uc3m | Universidad Carlos III de Madrid

Grado en Ingeniería Informática

2017-2018

Trabajo Fin de Grado

“Sistema genérico de metadatos basado en Zookeeper para sistemas de ficheros distribuidos y paralelos”

Volodymyr Fedorutsa

Tutor:

Félix García Carballeira

Lugar y fecha de presentación prevista

ABSTRACT

Actualmente existe una gran necesidad de realizar cómputo en la nube, ya sea por usuarios corrientes o especializados como ingenieros, científicos o investigadores. La presente memoria se corresponde con la información correspondiente al diseño e implementación de un sistema de ficheros distribuido y paralelo basado en ZooKeeper.

Este sistema de ficheros en esencia es una infraestructura que cubre la necesidad de adaptar una herramienta de sincronización de servidores como ZooKeeper, a uno de los estándares más conocidos: POSIX, para simplificar trabajos futuros en los campos de almacenamiento de información en la nube.

El *modus operandi* más básico, la idea principal, es que un usuario del sistema de ficheros puede organizar la información en particiones, donde la parte que va a usar más es la de ver información de directorios (o carpetas) y ficheros (o archivos). La otra parte corresponde con información respecto de cuántos servidores de datos hay, qué dirección IP tienen y cuál es el tamaño de bloque que van a manejar, pero esto solo se decide una vez, mientras que la información de directorios y ficheros se maneja frecuentemente. Todo este conjunto de datos se denominan metadatos y se almacenan en los servidores de ZooKeeper. Ahora bien, la información o datos que hay dentro de los ficheros se almacena en los servidores de datos. Este manejo de información es el caracteriza a nuestro sistema de ficheros como distribuido y paralelo.

Debido a que el manejo de metadatos es con ZooKeeper, existe cierta seguridad de que la eficiencia está garantizada, y el sistema soportará una carga de usuarios elevada. Asimismo, los servidores de datos tienen un algoritmo para distribuir los bloques de datos que consiste en elegir un servidor para el bloque inicial en base a una función resumen del nombre del fichero, y los bloques sucesivos se distribuyen siguiendo el algoritmo de Round Robin, garantizando una distribución equitativa de carga también en los servidores de datos.

Palabras clave: File server, File systems, Distributed Computing, Parallel file system, Distributed file systems.

AGRADECIMIENTOS

Gracias a mi madre por el apoyo.

Gracias a mi pareja y amigos por los ánimos.

Gracias a mi tutor Félix por los conocimientos y la guía.

Sin vosotros no hubiese sido posible, gracias a todos de corazón.

INTRODUCTION

Nowadays many online services make use of cloud computing, an obvious example of these services are Google Drive, Dropbox and OneDrive. But online services are not limited to just storage. Popular examples of online services are Spotify and Netflix, one provides online music streaming and the other online video streaming.

Now that the idea of an online service is clear, there is a need to expand upon this concept. These systems utilize distributed and parallel computing to archive their respective functionality. In the examples, the services are available to a wide public, but there is a lot of distributed systems that are not public.

Many fields like big data, scientific research and high-performance computing make use of distributed and parallel systems, which aids the research bringing every party together. This makes sharing and processing of information easier along different organizations.

Given the importance of the stated fields of interest and with the notion that these research fields need to process huge amounts of data (usually multiple petabytes), comes the motivation of expanding the existing knowledge in distributed and parallel computing.

The main purpose of this thesis is the development and implementation of a generic metadata system using ZooKeeper. One of the main requirements is that the interface of this system must be compliant to the POSIX standards.

Given that the developed metadata system is generic, it is possible to associate almost any data system along this metadata so that they work together. In this thesis, a simple socket-based data server has been developed as a proof of concept that it is possible for our metadata system to be coupled with a data system.

STATE OF THE ART

In this section, the main characteristics of large file systems will be discussed. We will point out some of the main flaws that these filesystems have and offer a tool which gives a solution to these difficulties. This tool is ZooKeeper, which is used to develop our generic metadata system.

First and foremost, let's talk about the parallel aspect of some filesystems. They are mainly used in high performance computing environments and have the purpose of processing huge amounts of data. Some file systems like Lustre have got a server dedicated for metadata processing that is non-overlapping with the data servers. Other file systems like PVFS have servers that keep the data and metadata in the same servers. While Lustre has got a native POSIX compliant interface, ZooKeeper has not got one. Furthermore, the most popular interface for ZooKeeper is written in Java. The goal of our work is to make use of the C API of ZooKeeper to build a POSIX interface. The parallel aspect of ZooKeeper is present given that it supports multiple clients at the same time, but it is not as powerful as other file system's like Lustre. However, the advantage that ZooKeeper has over Lustre, is that Zookeeper's operations are guaranteed to happen in the same order as they have been called. This means that even though the client calls happen in a distributed and parallel environment, ZooKeeper makes sure the order of calls stays the same and happen in sequential order.

Next, how metadata is treated in parallel file systems. Some traditional file systems like the original Network File System (NFS) are different from more modern parallel file systems like Lustre in the way they separate their metadata from their data. In NFS the servers must handle the data and the metadata. This implies an increment in the load the server experiences, thus reducing the performance and scalability of the file system. In contrast, parallel file systems store the metadata in a different server known as the metadata server. Even though this separation is made, it may not be sufficient in the HPC environments this parallel file systems is used. Given that the majority of have some metadata operations associated with them, only one metadata server sometimes falls short. If the metadata server fails, inevitably all the file system's operations that use metadata calls fail, making the file system almost useless.

One of the solution to this problem is to have more than one metadata server in a cluster, but this solution has its own problems. Using the clustered metadata servers, all the

different metadata servers can listen and process the clients calls, so the overall load is distributed along all the metadata servers. However, the main problems this brings is the error-recovery, the consistency of the metadata along the different metadata servers and the reliability of the file system. ZooKeeper is designed from its root to guarantee recovery from errors, metadata replication, atomic operations and overall system reliability. ZooKeeper guarantees that when a client updates some piece of information, that information stays the same until it is overwritten. Moreover, the user will always see the same version of the service, no matter to which metadata server it connects to in the cluster. Lastly, all the operations end up in a success or fail, there is not in-between or gray areas. All these factors make ZooKeeper a consistent infrastructure to build upon.

On a final note, the trend amongst file systems is to have a single metadata server. Some others have a backup metadata server in case the first one fails, but they never work at the same time to expand on the overall throughput. One of the main bottlenecks to increase parallel file system performance is update the current metadata management policies.

SOFTWARE ANALYSIS

In the process of searching for the best tools upon which it would be possible to build the generic metadata system, we found Apache Curator, etcd and Apache ZooKeeper and made the following table to compare the possible solutions:

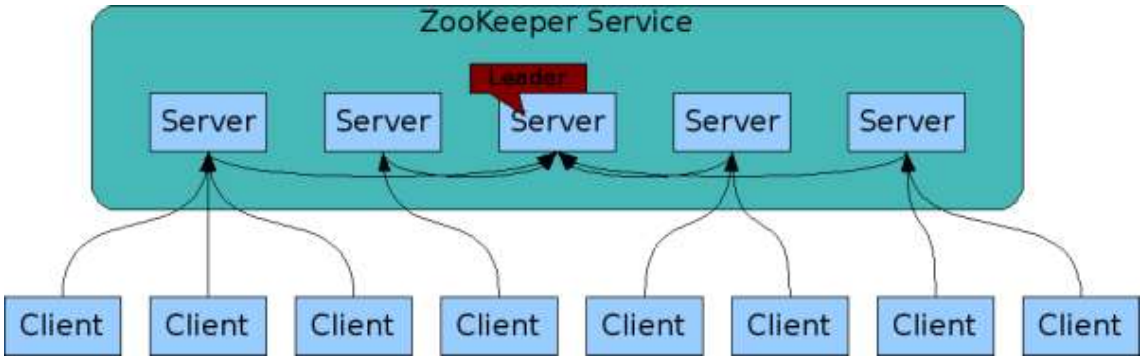
Tool name	Programming Language	Interface supports creation of files or directories	Documentation	Difficulty
Apache Curator	Java	Yes	Detailed	Medium/Low
etcd	Go	No	Detailed	Medium
Apache ZooKeeper	Java o C	Yes	Detailed	Medium

Even though etcd had good reviews, it is not programmed in C, nor gives such interface, nor supports creation of directories or files (nodes in general). By recommendation of my

tutor and basically because ZooKeeper is the only tool that supports a C interface, it was the chosen tool.

ZOOKEEPER

ZooKeeper is a tool that serves as a centralized service to keep metadata synchronized along multiple servers that make a cluster. The following illustration is a good demonstration of how the connections happen between the clients and servers.



As we can see, the ZooKeeper service has a Leader which only function is to synchronize all data between the servers. The leader is also responsible of making sure there is some degree of redundancy in the system. Redundancy means copy blocks of information and store copies in different servers to make it fail-proof. Even if one of the servers goes down, all its information is still available.

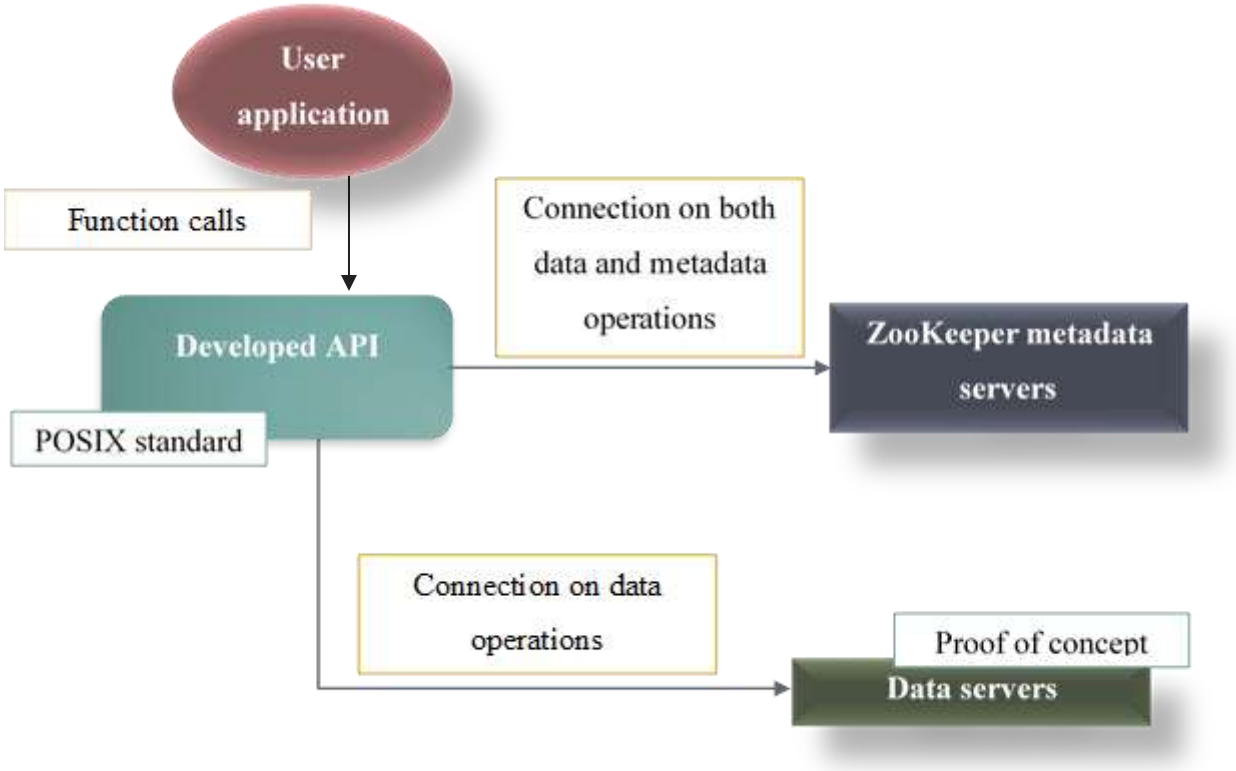
In the situation that the leader of the ZooKeeper service fails, there is a policy for a new leader election. The protocol accomplishes this task consists in the following:

- Every server creates a znode with the name “n_” and the flags of EPHEMERAL and SEQUENTIAL inside a directory named, for example, “/election”.
- The server that has the lowest znode sequential number is the leader.
- There are some periodical checks performed to check if the leader is still operative. In case the leader server is down, the server that had created the znode with the next lowest number is elected the new leader.

There have been multiple projects built upon ZooKeeper due to this fail-proof system and other advantages discussed in the state of the art section. Most notably, it is used by Yahoo! and HDFS filesystem, which are two of the most popular implementations of ZooKeeper.

DESIGN

In the following illustration, we draw a diagram to portray the communications flow and relation between all the developed components. We also explain what the meaning of each of the elements is so that a solid idea is formed about the file system.



- User application: The interface to which the user connects. With this application the user can operate with the metadata service and with the data through the developed API. This piece of software is also known as the client application. It accesses one or multiple servers through commands entered by the user, and it works like a black box for the user.
- Developed API: These are functions that are made in POSIX standard where most of these functions make use of the ZooKeeper API, along some additional configuration code to ensure proper functionality. ZooKeeper calls provide the interaction with the ZooKeeper metadata servers, meanwhile read and write operations provide interaction with the data servers.
- Metadata servers: These are the servers that handle the client metadata that consists of information about the node proprietary, permissions, creation and last modification times amongst others.

- Data servers: Developed as a proof of concept to prove that the metadata system can work alongside some data server. In this position you can insert any data server of your preference. This unit provides the handling of data corresponding to the files stored in the metadata system.

This design is aimed to be modular, versatile and generic. This means that if you wish to connect the client with an existing system, you can check if the calls are compatible and plug it in. If you want to replace the existing data server with another one of your choice, maybe some adaptations are required to make it work, but it should work with almost no effort.

Metadata structure

The purpose of this subsection is to explain the information that is stored in the metadata system for each file or directory that is stored in the metadata system. This represents the interaction between the developed API that defines the content stored in each node, and the ZooKeeper metadata servers, that store the information.

For the design of this structures, we started to analyze the traditional file systems like the ones that are present in the operating systems of our personal computers. This was the starting point so that the client would have little to no difficulty in using our file system. The difference to keep in mind is that the developed file system must manage data from multiple users at the same time. Thus, there must be an attribute that stores the owned id in each node.

Alongside the owned id, it is convenient to store the timestamp in which the file has been created and another one for the last modification time, this information is useful to the users. The type of node is very important. In our POSIX compliant metadata system there is files and directories, while in ZooKeeper there is only nodes, that can contain other nodes inside of them.

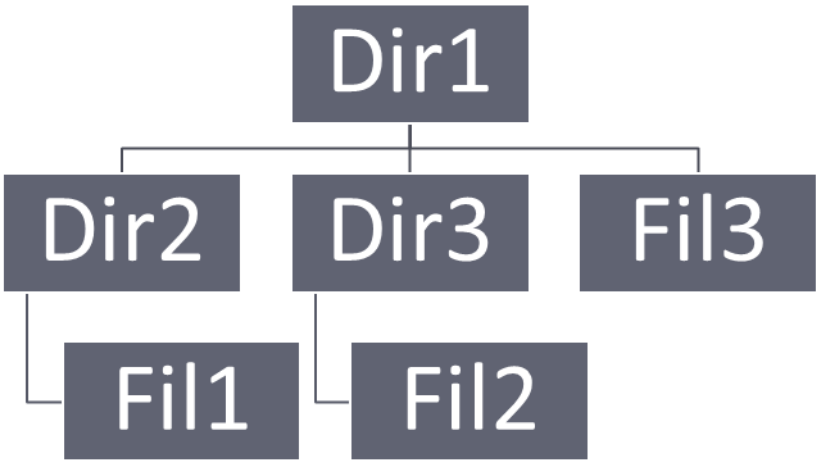
To guarantee the privacy of each file and directory created by a user, we have a set of access permissions for the creator of the file. The size of the file or directory is also important. While the size of the file is given by the amount of information it stores, the size of a directory is the sum of the size of all its child nodes.

The last piece of information both directories and files have in common is the complete path to the node. This is useful to locate any node in the metadata system. Traditional file

systems tend to store this information too, as there is a need to locate any file from the root directory.

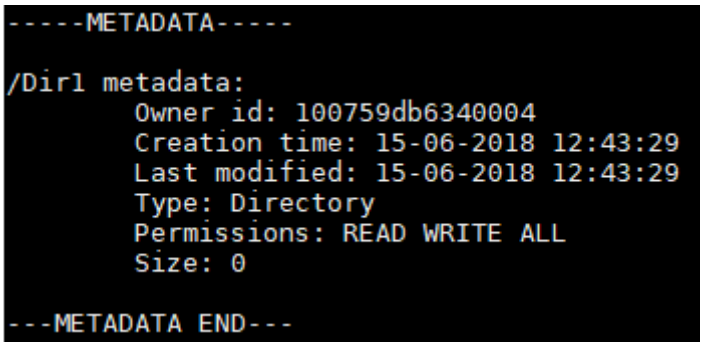
The following information is reserved for files only. In first place there is a piece of information that stores the server on which the first block of data is stored. Later we will discuss as to why this is important.

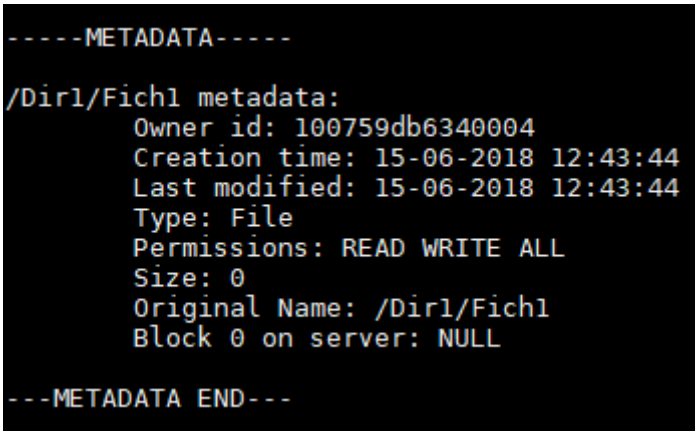
In second place, the original name of the file is also stored. This means that even if a file is renamed it keeps its original name forever. This is a solution we implemented to circumvent a problem we had with the renaming of the nodes. As it seems, ZooKeeper does not allow a native renaming of the nodes, thus we implemented a renaming system that deletes a node and creates it with the new name (preserving the previous metadata). The problem is that this only works with leaf nodes.



In the previous illustration Fil1, Fil2 and Fil3 are the leaf nodes (they do not have any child nodes). So, why do we store the original name? To keep track of the data of the file in the data servers, as the file data is stored in the same path it is in the metadata servers. Once the data is stored in the data servers and the file name changes in the metadata, it would be time consuming to also rename all the data blocks associated with the file.

To demonstrate how a directory “Dir1” and a File “Fich1” which is inside Dir1 metadata look, we made two pictures to illustrate. The information seen in the following pictures is the corresponding information a user would see if he called the function zstat of the developed API onto any node:





Data server structure

It has two main functions:

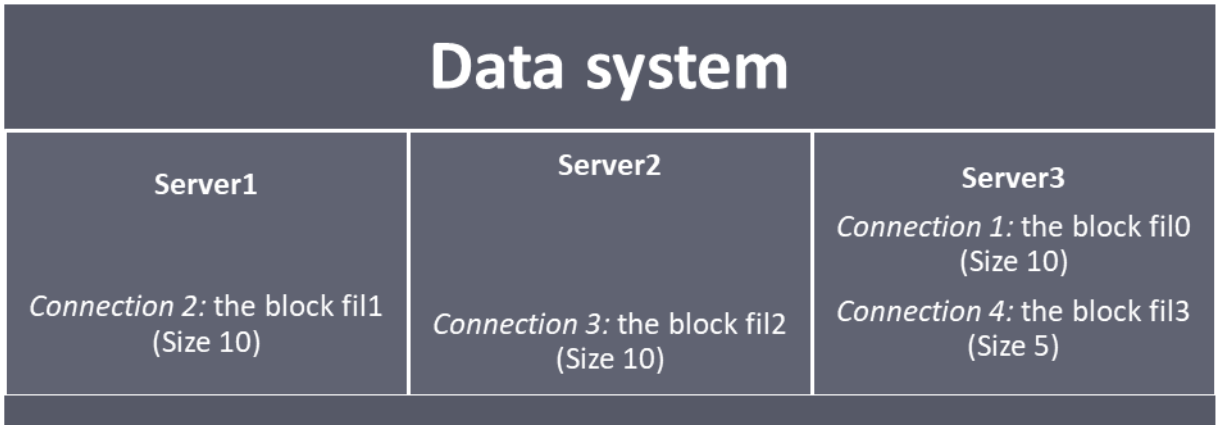
- PUT_BLOCK, compliant to the following protocol:
  - Waits to get the file path
  - Waits to get the data block size to be written
  - Waits to get the data block
  - Waits to get the pointer position to the current data block
  - Write the data block in the path
  - Return number of bytes written
- GET_BLOCK, compliant to the following protocol:
  - Waits to get the file path
  - Waits to get the data block size to be read
  - Waits to get the pointer position to the current data block
  - Reads the corresponding data block
  - Return the read data block

The developed API follow the complementary protocol, sending each one of the pieces of information to the data servers and receiving the data they return. We can establish a correlation between PUT_BLOCK and GET_BLOCK, as our zwrite function makes the call to PUT_BLOCK and zread function makes the call to GET_BLOCK.

Client access to the data

From a client's perspective, the system is simple. The client must indicate the file to which he wants to write and the data it wants to be stored. The rest of the operations are handled by the API and servers and are not known to the user.

The process is best understood with an example. Let's imagine a client creates a partition with blocksize 10 bytes and 3 data servers. The client wishes to write 35 characters into the file named "fil", this is what happens underneath:



The read operations happen in the same way, the number of bytes to be read are divided into several blocks depending on the blocksize and size of the file. Then connections are made based in the number of blocks that need to be extracted from the data servers.

CONCLUSIONS

The development of the generic metadata system with the aid of ZooKeeper, coupled with a proof of concept data server has ended up being a success. Only one piece of the functionality has been reduced from the initial planning, this functionality being the renaming of the nodes. As we mentioned in the design, ZooKeeper does not support native renaming of the nodes and we did not know at the time but managed to make a simple renaming system.

Regarding other design goals, the goal of developing a metadata system that complies to the POSIX standard has been proved a success. The client that we developed can also use (or consume from) the interface provided by the developed API. The system also works with a coupled data server. All the components have been tested and proven to work together.

Without a doubt, ZooKeeper has been proven to be very useful in this task, I would not hesitate to use ZooKeeper in future projects that suit its needs. Moreover, we did not explore all of the available functions that ZooKeeper offers like the “watcher” functionality, so that would be interesting to explore in the future.

ÍNDICE DE CONTENIDO

ÍNDICE DE ILUSTRACIONES

ÍNDICE DE TABLAS

1. INTRODUCCIÓN

Este capítulo consiste en la explicación del porqué de este proyecto, concretamente la motivación por la cual se obtiene la idea, la descripción de los objetivos de este trabajo, el alcance de las tareas, los destinatarios a los que va dirigido el documento y la estructura en la que consistirá la presente memoria.

1.1. Motivación

Actualmente se utilizan cantidad de servicios en la nube, un ejemplo de estos servicios obvios para una persona corriente puede ser Google Drive o sus variaciones como Dropbox o OneDrive, pero los servicios en la nube no están limitados a almacenamiento de ficheros de uso personal. Dos ejemplos de servicios en la nube que se diferencian y aun así son muy populares podrían ser Spotify y Netflix que ofrecen un servicio de música y vídeo respectivamente.

Estos servicios han tomado popularidad ya que a las personas normalmente no les interesa comprar todas películas que quieren ver, o todas las canciones que quieren escuchar y Netflix o Spotify conceden una amplia colección de contenido a través de internet por una cuota mensual más cómoda. Existen incluso servicios en la nube que permiten jugar a videojuegos sin que el videojuego en cuestión se esté ejecutando en la máquina donde se juegue, un ejemplo de esto es Nvidia GameStream para Nvidia SHIELD.

El uso de sistemas distribuidos y paralelos no se limita a ofrecer servicios a usuarios, sino que también es significativo en campos como Big Data que investiga una solución a las cantidades ingentes de información que tienen que manejar los sistemas informáticos hoy en día. Sectores como la salud pública con registros de enfermedades y alergias, incluso órganos como el gobierno con certificados de nacimiento e información civil son ejemplos de sectores en los que abunda la información. Big Data también es utilizado en el campo de la ciencia como por ejemplo el famoso gran colisionador de hadrones (partículas) que fue noticia hace unos años.

En grandes organizaciones se manejan cantidades de datos de más de un petabyte en sus centros de datos principales y en muchas de estas organizaciones existe una replicación de datos en tres o más lugares, por lo que es posible imaginar que el tamaño de datos se triplica.

De aquí surge la motivación de ampliar conocimientos sobre cómo funcionarían estas tecnologías ya que están en auge y cada vez se ven más servicios distribuidos de todo lo imaginable.

Respecto de las cuestiones técnicas de estos servicios distribuidos, es interesante pensar en cuál es el número de servidores que hace falta para manejar las peticiones de sus usuarios, así como cuántos servidores hacen falta para almacenar el contenido que maneja cada servicio. También surge la duda de si se podría mejorar la eficiencia de estos servicios cambiando algún concepto como el tamaño de bloque que utilizan o los algoritmos de distribución de bloques, así como el grado de replicación que tienen.

El conjunto de las cuestiones de expansión y conceptos técnicos son las que hacen el tema de este trabajo interesante y relevante para el futuro de estas tecnologías.

1.2. Objetivo

El objetivo principal es el desarrollo e implementación de un sistema genérico de metadatos utilizando ZooKeeper. La finalidad es que el sistema de metadatos esté hecho de tal forma que la interacción con él es igual a la de un sistema de ficheros en estándar POSIX.

Para garantizar que el acceso es distribuido un servicio de metadatos de un sistema de ficheros puede ejecutarse en varias máquinas remotas, mientras que para garantizar que el acceso es paralelo, está garantizado por el uso de ZooKeeper.

Ya que es un sistema genérico de metadatos, significa que es posible asociar prácticamente cualquier sistema de datos a él, como prueba de concepto se ha desarrollado un sistema de datos propio, que también garantiza un acceso distribuido mediante la utilización de varios servidores de datos y paralelo porque las consultas se atienden mediante hilos (*threads*), lo que permite escuchar peticiones de varios usuarios al mismo tiempo.

1.3. Alcance

Existen una serie de etapas de desarrollo diferenciadas del proyecto:

- Desarrollar el servicio de metadatos.
- Proporcionar una API en estándar POSIX para un cliente del servicio de metadatos.

- Desarrollar un servidor de datos para dar soporte a las funciones que tienen que ver con manejo de archivos del sistema de ficheros (escritura, lectura).
- Añadir a la API unos métodos correspondientes a sistemas de ficheros en estándar POSIX para el servidor de datos.
- Implementar el servidor de metadatos y los servidores de datos, así como garantizar que funcionan en conjunto.
- Desarrollar una librería del lado del cliente, para que pueda utilizar el servicio de metadatos y datos.

1.4. Destinatarios

El presente proyecto está dirigido a todo usuario técnico que quiera utilizar la API desarrollada para comprender al detalle su funcionamiento, así como para cualquiera que quiera ampliar las capacidades existentes del sistema de ficheros

1.5. Estructura del documento

La estructura del documento es la siguiente:

- **Introducción**: Se describe el porqué del proyecto y características del presente documento.
- **Estado de la cuestión**: Se explica la situación actual en el uso de la herramienta ZooKeeper, así como el impacto que tiene esta tecnología y qué proyectos se han hecho con ella.
- **Análisis**: Apartado en el que se detallan los requisitos que se contemplaban para el sistema.
- **Diseño**: Se describe cuál es la arquitectura del sistema de ficheros y sistema de almacenamiento.
- **Implementación e implantación**: Aquí se detallan los puntos más complicados del desarrollo y se indica cuál es el proceso de despliegue.
- **Evaluación**: Se evalúa la eficiencia del sistema genérico de datos y del servidor de datos.
- **Planificación y presupuesto**: Sección dedicada a describir la planificación y medir el tiempo que ha llevado la realización del proyecto, así como establecer el impacto de ese tiempo en el presupuesto.
- **Conclusiones**: Se detallan las ideas más claras que se extraen después de realizar el proyecto.

- **Apéndices**: Apartado donde se incluye un manual de instalación detallado.
- **Bibliografía**: Referencias del trabajo.

2. ESTADO DE LA CUESTIÓN

En este capítulo se discutirán los métodos relacionados con el manejo de metadatos en entornos de alto rendimiento computacional. Se destacan los defectos de las implementaciones de la mayor parte de sistemas de ficheros paralelos y se presenta la herramienta ZooKeeper que soluciona dichos problemas.

2.1. Sistemas de ficheros paralelos

Los sistemas de ficheros paralelos se utilizan mayoritariamente en entornos donde se necesita un alto rendimiento computacional, en los cuales se maneja un gran volumen de datos. Algunos sistemas de ficheros como Lustre [1] tienen un servidor de metadatos a parte de los servidores de datos, pero otros sistemas de ficheros paralelos como PVFS [2] mantienen los datos y los metadatos en el mismo lugar o servidor. Lustre es un sistema de ficheros de código abierto que obedece el estándar POSIX. Debido a la arquitectura extremadamente escalable que tiene, es muy popular en ámbitos científicos donde se tienen supercomputadores, así como en otros ámbitos como el sector financiero. ZooKeeper no ofrece una interfaz POSIX de forma nativa, y de hecho su distribución más popular está escrita en Java, pero la ocupación de este trabajo es obedecer un estándar POSIX a través de la API de ZooKeeper que está escrita en C. La paralelización en ZooKeeper se produce de forma simbólica, por el hecho de que se permiten múltiples conexiones de distintos clientes al mismo tiempo. Aun así, las operaciones de escritura pasan por el servidor líder y ZooKeeper garantiza el orden de las operaciones, así que en esencia se producen de forma secuencial.

2.2. Manejo de metadatos en sistemas de ficheros paralelos

Sistemas de ficheros paralelos como Lustre se diferencian de los sistemas de ficheros distribuidos clásicos como NFS [3] en la forma en la que separan el manejo de metadatos de los datos. En los sistemas de ficheros clásicos como NFS el servidor tiene que manejar tanto datos como metadatos. Esto causa un incremento en la carga del servidor y limita el rendimiento y la escalabilidad del sistema de ficheros. Por otro lado, los sistemas de ficheros paralelos almacenan los metadatos en un servidor a parte conocido como el servidor de metadatos.

Aun así, la mayoría de los sistemas de ficheros paralelos tienen un único servidor de metadatos [4]. En un supuesto momento en el que la carga del servidor de metadatos incremente, el rendimiento de dicho servidor de metadatos inevitablemente decrementa,

reduciendo el rendimiento de todo el sistema de ficheros (y en ocasiones imposibilitando su funcionamiento).

Una de las soluciones propuestas es utilizar un conjunto de servidores de metadatos en un clúster, pero esto también tiene una serie de inconvenientes. Con el clúster de metadatos, básicamente distintos servidores de metadatos pueden atender las peticiones de múltiples clientes y la carga se distribuye. Las dificultades más notables que trae un clúster de servidores de metadatos es la recuperación en casos de error, la consistencia de los metadatos a través de los distintos servidores y la confiabilidad general del sistema. ZooKeeper está diseñado desde un principio para proporcionar un acceso ordenado y atómico. También garantiza la confiabilidad ya que una vez se haya aplicado una actualización, persistirá hasta que un cliente sobrescriba dicha actualización. Además, un cliente siempre verá la misma versión del servicio sin importar a qué servidor se conecte de todos los disponibles en el clúster. Por último, todas las operaciones terminan en éxito o fallo, no existen resultados mixtos ni parciales. Todo este conjunto de factores hace a ZooKeeper una infraestructura consistente sobre la que construir un sistema de metadatos.

2.3. Servicio de coordinación distribuido

Los mecanismos de gestión de cerrojos existen en todo ordenador que es parte de un clúster y sirven para garantizar un acceso distribuido y sincronizado a recursos compartidos. Un ejemplo muy reconocido de esto es Chubby [5], desarrollado por Google. Entre los principales objetivos de Chubby se encuentra la confiabilidad, disponibilidad para un amplio conjunto de clientes y una interfaz fácil de entender, la capacidad de almacenamiento y el ancho de banda están en segundo plano de prioridad. La interfaz de Chubby es similar a la de un sistema de ficheros sencillo que ejecuta lecturas y escrituras de ficheros enteros, mejorado con cerrojos y notificaciones de eventos como la modificación de un archivo. Chubby sirve de ayuda a los desarrolladores para lidiar con cuestiones como la sincronización entre sus sistemas y, más concretamente, con la elección de un líder entre servidores equivalentes.

Apache ZooKeeper es un clon de Chubby diseñado para satisfacer muchos de los mismos objetivos que Chubby para sistemas de ficheros HDFS [6] y otras infraestructuras de Hadoop. ZooKeeper es un servicio de coordinación para aplicaciones distribuidas. Ofrece una serie de interfaces que las aplicaciones distribuidas pueden aprovechar y construir sobre ellas para implementar niveles más sofisticados de sincronización, configuración y

gestión del espacio de nombres. ZooKeeper permite la sincronización entre procesos distribuidos a través de un espacio de nombre jerárquico compartido, que está organizado de forma similar a un sistema de ficheros corriente. El espacio de nombre se compone de nodos característicos de ZooKeeper que ellos llaman Znodos. Los Znodos no almacenan datos sino información de configuración (o metadatos). Como dato diferenciador, ZooKeeper tiene un mejor rendimiento cuando las operaciones predominantes son de lectura frente a escritura en un ratio de aproximadamente 10 operaciones de lectura a 1 de escritura. [7]

2.4. Métodos de manejo de metadatos

El manejo de metadatos en sistemas de ficheros ha sido un área de investigación activa desde hace tiempo. Con la llegada de los sistemas de ficheros paralelos el manejo de metadatos de forma eficiente y escalable se ha convertido en una prioridad y es una tarea complicada. Los sistemas de ficheros distribuidos dedican frecuentemente un subconjunto de servidores para manejo de metadatos. Mapear las semánticas de datos y metadatos a través de distintos servidores no superpuestos permite a los sistemas de ficheros ser escalables en términos de rendimiento I/O (entrada/salida), así como capacidad de almacenamiento.

Sistemas de ficheros como NFS, AFS [8], Lustre y GFS [9] utilizan un único servidor de metadatos para manejar globalmente el espacio de nombres de un sistema de ficheros compartido. Aunque el diseño es sencillo, no es escalable, y resulta en el servidor de metadatos siendo un cuello de botella; además de ser peligroso porque si falla este único servidor de metadatos, cesa el funcionamiento de todo el sistema de ficheros distribuido. Los sistemas de ficheros como NFS y AFS son capaces de particionar su espacio de nombres de forma estática a lo largo de múltiples servidores, así que la mayor parte de las operaciones de metadatos son centralizadas.

En resumen, se observa un patrón y es que el servidor de metadatos suele ser único y como mucho existe un servidor adicional por si falla el primero, pero éste último no sirve para aumentar el ancho de banda. En otras palabras, uno de los impedimentos principales para aumentar el rendimiento de los sistemas de ficheros paralelos son las políticas actuales de manejo de metadatos.

2.5. ZooKeeper

ZooKeeper es un proyecto de la reconocida fundación (sin ánimo de lucro) Apache Software Foundation, cuyo software más reconocido es el servidor HTTP Apache. ZooKeeper es una herramienta que en esencia es un servicio centralizado para mantener metadatos sincronizados a lo largo de varios nodos (servidores) que componen un clúster (o conjunto de servidores que cumplen una misma función).

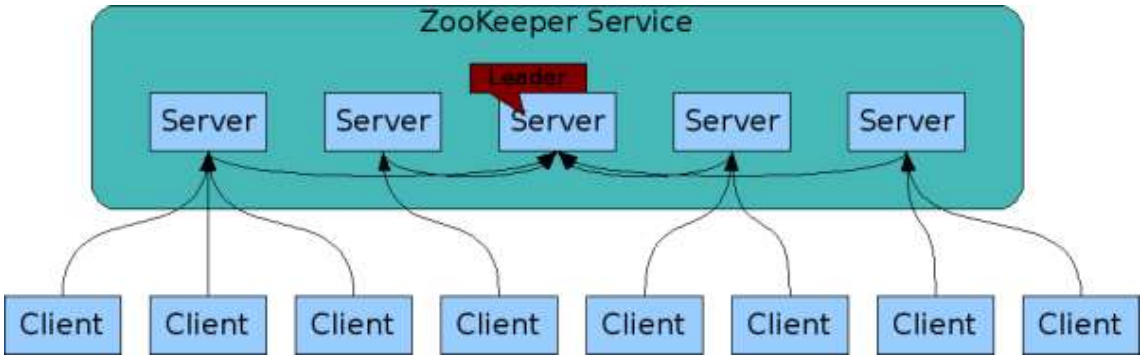


Ilustración 2-1. Arquitectura de ZooKeeper

Como se puede observar en la ilustración, el servicio de ZooKeeper tiene un líder, que es una forma de decir que existe un servidor que sincroniza a todos en términos de sincronizar la información y garantizar redundancia, la redundancia es copiar bloques de datos y almacenar dichas copias en un servidor distinto del original. La redundancia garantiza que al fallar o apagarse un servidor, se tenga un repuesto para permitir que el servicio siga en ejecución.

En caso de que el servidor que haya fallado sea el líder hay que establecer, mediante una política de elección de líder, un nuevo servidor. ZooKeeper maneja esto de la siguiente forma:

- Los servidores crean un znode con nombre n_ y flags de efímero y secuencial en una ruta como por ejemplo /election.
- El servidor que haya creado el znode con un número de secuencia más pequeño se convierte en líder.
- Se comprueba de forma periódica si el líder sigue estando ahí, en caso de que no lo esté el servidor que haya creado el znode con el número de secuencia más bajo pasará a ser el líder.

Debido a esta forma de funcionamiento, ZooKeeper proporciona un ancho de banda mayor a los sistemas de ficheros comentados en el punto 2.1, así como asegura la consistencia y disponibilidad de los metadatos en gran medida sobre los sistemas anteriores.

Se han desarrollado otros proyectos a partir de ZooKeeper, algunos buscan simplificar su uso a un lenguaje de más alto nivel y ampliar su funcionalidad como es el caso de Apache Curator que es una librería de cliente para ZooKeeper basada en Java y otra librería que pretende hacer el uso de ZooKeeper más fácil y libre de errores con Python llamada Kazoo.

ZooKeeper también es utilizado en proyectos de software libre como Eclipse Communication Framework, en diversos proyectos de Apache como MapReduce e incluso en compañías como Yahoo! y Zynga [10]. En resumen, es una herramienta que actualmente se utiliza mucho y en una variedad de proyectos, lo que caracteriza a ZooKeeper como útil, fiable, versátil y eficiente.

2.6. Impacto socioeconómico

En una aplicación tan reconocida como Netflix, se utiliza la librería de cliente Apache Curator que es en esencia ZooKeeper [11]. Aplicaciones como Netflix son muy utilizadas por las personas como se puede ver en la siguiente gráfica:

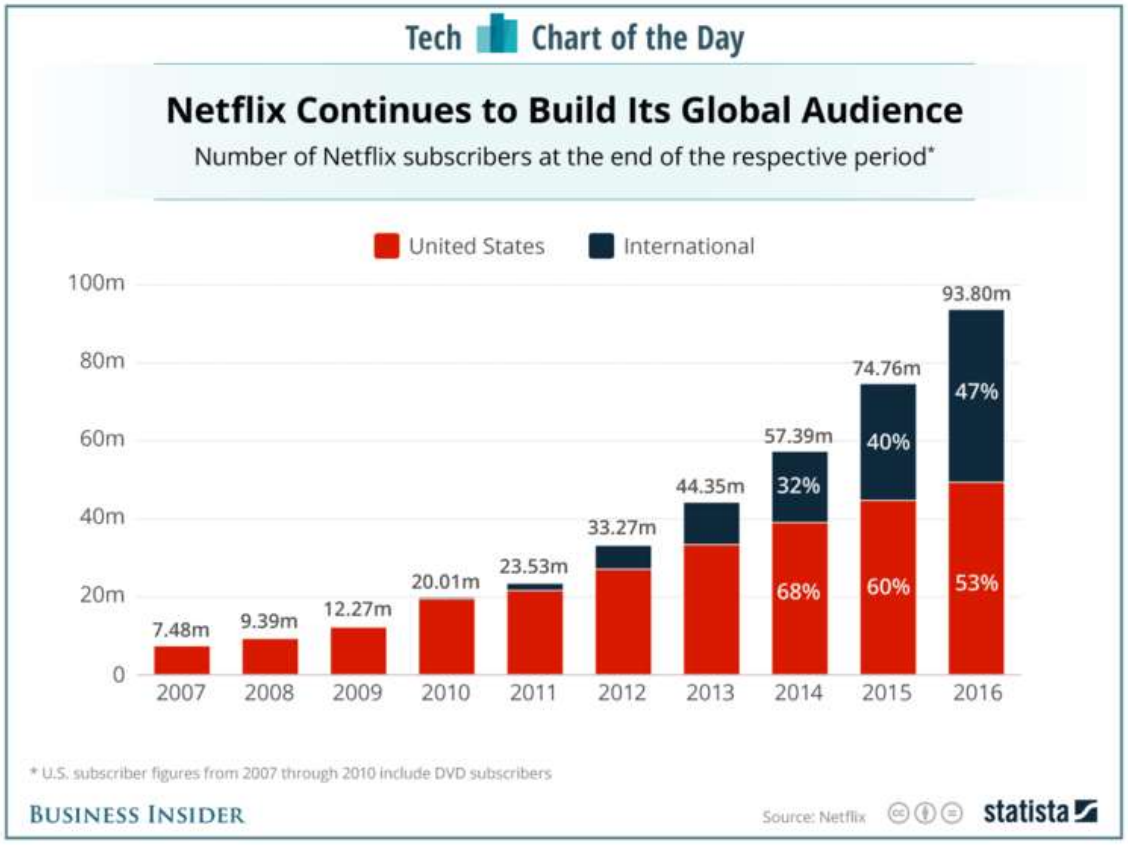


Ilustración 2-2. Suscripciones globales a Netflix

Es posible estimar que en el año 2018 hay cerca de 100 millones de suscripciones globales a Netflix, siendo la suscripción estándar 11 euros al mes esto equivale a 1,1 billones de euros al mes y esto solo es Netflix, hay más aplicaciones como Spotify que también es muy conocida y también ofrece un servicio distribuido, aunque no utilice ZooKeeper concretamente debe de utilizar alguna tecnología similar.

Si herramientas como ZooKeeper son necesarias en aplicaciones como estas que ofrecen un servicio legítimo por el que muchas personas están dispuestas a pagar una cuota mensual, entonces queda demostrada la importancia de dichas herramientas. Más aún si se reflexiona en que mejoras de eficiencia en estas herramientas podrían suponer una reducción en el coste de las suscripciones mensuales o el coste de los productos que utilicen estas herramientas.

2.7. Trabajos Similares

No se han encontrado trabajos que hagan un sistema de ficheros a partir de ZooKeeper porque el objetivo principal de ZooKeeper no es ayudar a implementar sistema de ficheros sino manejar servidores en un clúster.

Los principales problemas de usar ZooKeeper como sistemas de ficheros son su capacidad de manejar grandes volúmenes de datos y el acceso a dichos datos, pero en este proyecto esto se solventa almacenando en ZooKeeper únicamente metadatos (que no llegan a 1 Megabyte de tamaño, lo que garantiza la eficiencia de ZooKeeper), mientras que los datos se almacenan en unos servidores de datos a parte diseñados e implementados por separado.

Existen trabajos previos hechos sobre ZooKeeper, pero no se ha encontrado ninguno que utilice su API de C, además que los proyectos son de otros ámbitos que no son sistemas de ficheros, como ya se comentó al final del punto 2.

Cabe destacar que, como se ha comentado en el apartado de estado del arte, el sistema de ficheros Lustre en alguna de sus versiones proporciona un clúster de servidores de metadatos y su interfaz obedece al estándar POSIX. Aun así, el propósito de Lustre es ofrecer un sistema de ficheros completo para entornos de alto rendimiento computacional. Por otra parte, el objetivo de este proyecto es proporcionar un sistema genérico de metadatos al que se pueda acoplar cualquier sistema de datos y que sirva de infraestructura para construir sistemas más complejos sobre sobre él.

3. ANÁLISIS

Este apartado está destinado al marco regulador, comparar las distintas herramientas posibles que había para realizar el sistema de ficheros y a los requisitos del sistema.

3.1. Comparación de soluciones

En la búsqueda de herramientas que se podrían haber utilizado, se han encontrado algunas que destacaban sobre el resto en popularidad, pero se busca que su lenguaje sea C, que tenga la capacidad de crear directorios y archivos, que tenga una buena documentación y en esencia sea relativamente sencillo trabajar con dicha herramienta.

Herramienta	Lenguaje	Creación de directorios/archivos	Documentación	Dificultad
Apache Curator	Java	Si	Extensiva	Media/Baja
etcd	Go	No	Extensiva	Media
Apache ZooKeeper	Java o C	Si	Extensiva	Media

Tabla 1. Comparación de soluciones

Se hablaba bien de etcd, pero utiliza un lenguaje de programación que no nos conviene, además de que no es posible crear una serie de directorios ni ficheros como es el caso de las otras herramientas. Apache Curator en realidad sería una herramienta ideal ya que es una ampliación de funcionalidad y simplificación sobre ZooKeeper, pero sólo sobre su versión Java. Por tanto, nuestra única alternativa, aunque no por ello la peor fue Apache ZooKeeper.

3.2. Marco regulador

Respecto de los aspectos técnicos y legales, la única normativa por la que se rige el presente trabajo es la de Creative Commons Reconocimiento – No Comercial – Sin Obra Derivada tal y como se indica en la portada del trabajo.

ZooKeeper es la principal herramienta utilizada para el desarrollo del sistema de ficheros y es un proyecto voluntario y open source (de código abierto) [12]. El trabajo se desarrolló en una distribución Linux basada en Ubuntu, cuya licencia es GNU GPL [13] que permite a los usuarios finales estudiar, compartir y modificar el software.

3.3. Requisitos

Según el estándar IEEE 830 para especificación de requisitos software, las prácticas recomendadas para especificar requisitos son mediante requisitos funcionales y no funcionales con la posibilidad de incluir casos de uso.

La tabla que se va a utilizar para definir requisitos es:

ID	Identificador del requisito
Título	Definición breve del requisito
Descripción	Definición completa del requisito
Tipo	Funcional / No funcional
Importancia	Alta | Media | Baja
Pruebas de verificación	Comprobaciones del requisito

Tabla 2. Ejemplo de tabla de requisitos

3.3.1. Requisitos Funcionales

ID	RF-01
Título	Comando crear partición
Descripción	En el sistema de ficheros se deben de poder hacer particiones.
Tipo	Funcional
Importancia	Alta
Pruebas de verificación	Ejecutar el comando correspondiente a la creación de la partición y observar que la creación fue exitosa.

Tabla 3. RF-01

ID	RF-02
Título	Comando añadir servidor
Descripción	En el sistema de ficheros se deben de poder añadir servidores a las particiones a la ruta /nombreParticion/servidores, de forma que el conjunto de servidores de una partición maneje sus datos correspondientes.

Tipo	Funcional
Importancia	Alta
Pruebas de verificación	- Ejecutar el comando de añadir un servidor a una partición y observar que se añade y que se añade a la partición correcta - Asegurarse que los datos de una partición se manejan con los servidores indicados para esa partición

Tabla 4. RF-02

ID	**RF-03**
Título	Tamaño de bloque por partición
Descripción	En cada una de las particiones del sistema de ficheros es obligatorio almacenar un tamaño de bloque definido en los metadatos de un fichero situado en la ruta /nombreParticion/servidores/blocksize.
Tipo	Funcional
Importancia	Alta
Pruebas de verificación	- Asegurarse de que al ejecutar el comando de crear partición se crea el fichero - Asegurarse que en el momento de la creación tiene los metadatos correspondientes al tamaño de fichero

Tabla 5. RF-03

ID	**RF-04**
Título	Almacén de metadatos
Descripción	Al ejecutar el comando de crear partición, se deberá de crear la ruta /nombreParticion/metadatos/
Tipo	Funcional
Importancia	Baja
Pruebas de verificación	Asegurarse que al ejecutar el comando de crear partición se crea el directorio de metadatos.

Tabla 6. RF-04

ID	RF-05
Título	Metadatos del directorio servidores
Descripción	El directorio servidores de cada partición debe tener unos metadatos que indiquen el número de servidores que hay en su interior.
Tipo	Funcional
Importancia	Baja
Pruebas de verificación	- Asegurarse de que al ejecutar el comando de crear partición los metadatos iniciales de /nombreParticion/servidores contiene 0 servidores - Asegurarse de que al ejecutar el comando para añadir un servidor el número de servidores de la ruta /nombreParticion/servidores aumenta en 1

Tabla 7. RF-05

ID	RF-06
Título	Metadatos de directorios
Descripción	Al crear un directorio se le asignarán unos metadatos al directorio que constan de: - ID del usuario que creó el directorio - La fecha y hora de creación en formato: dd-MM-yyyy HH:mm:ss - La fecha y hora de última modificación en formato: dd-MM-yyyy HH:mm:ss - El tipo de nodo (en este caso directorio) - Permisos - Tamaño, el tamaño de un directorio está dado por el tamaño de los ficheros que contiene el directorio
Tipo	Funcional
Importancia	Alta
Pruebas de verificación	Asegurarse que al ejecutar el comando de crear directorio se le asocian los metadatos descritos

Tabla 8. RF-06

ID	RF-07
Título	Metadatos de ficheros
Descripción	Al crear un fichero se le asignarán unos metadatos al fichero que constan de: - ID del usuario que creó el directorio - La fecha y hora de creación en formato: dd-MM-yyyy HH:mm:ss - La fecha y hora de última modificación en formato: dd-MM-yyyy HH:mm:ss - El tipo de nodo (en este caso fichero) - Permisos - Tamaño, el tamaño de un directorio está dado por el tamaño de los ficheros que contiene el directorio - Nombre original, en caso de renombramiento del fichero, este nombre siempre se mantiene igual para que hagan uso de él los servidores de almacenamiento de datos - En qué servidor está situado el primer bloque de datos de este fichero, se indicará con un número de 0 al número de servidores donde 0 es el primer servidor creado en la partición, el valor inicial será NULL
Tipo	Funcional
Importancia	Alta
Pruebas de verificación	Asegurarse que al ejecutar el comando de crear fichero o abrir fichero con flag de O_CREAT se le asocian los metadatos descritos

Tabla 9. RF-07

ID	RF-08
Título	Descriptores de fichero
Descripción	Se debe de generar una estructura donde se mantengan en memoria los ficheros abiertos, así como los siguientes datos de cada fichero: - Número de descriptor de fichero - Posición del puntero de lectura/escritura - Nombre del directorio padre

	- Ruta completa - Permisos: 0 si solo lectura, 1 si solo escritura, 2 si se permite lectura y escritura - Tamaño del archivo
Tipo	Funcional
Importancia	Alta
Pruebas de verificación	Asegurarse de que cuando se abre un fichero se rellena una de las estructuras de descriptor de fichero.

Tabla 10. RF-08

ID	**RF-09**
Título	Función open
Descripción	La función debe de poder abrir o crear un archivo.
Tipo	Funcional
Importancia	Alta
Pruebas de verificación	- Al ejecutar la función con el flag O_CREAT, si no existe el archivo se creará - Al ejecutar la función sin el flag O_CREAT, si no existe el archivo será error - Al ejecutar la función sobre un directorio será error - Asegurarse de que, si se abre un fichero en modo lectura, no se puede escribir - Asegurarse de que, si se abre un fichero en modo escritura, no se puede leer - Comprobar que si se abre un archivo con flag de O_TRUNC se escribirá desde el comienzo del fichero - Comprobar que si se abre un archivo con flag de O_APPEND se escribirá desde donde se dejó de leer o escribir previamente - Comprobar que la función devuelve un descriptor de fichero válido

Tabla 11. RF-09

ID	RF-10
Título	Función creat
Descripción	La función debe de poder crear un archivo.
Tipo	Funcional
Importancia	Baja
Pruebas de verificación	Asegurarse de que, si no existía el archivo, se crea de forma satisfactoria y que en caso contrario se devuelve un error.

Tabla 12. RF-10

ID	RF-11
Título	Función close
Descripción	La función debe de poder cerrar un archivo que se abrió por la función open.
Tipo	Funcional
Importancia	Alta
Pruebas de verificación	- Asegurarse de que, si estaba abierto el archivo, se puede cerrar de forma satisfactoria con la función - Si se intenta cerrar un descriptor de fichero fuera de rango devolverá error - Si se intenta cerrar un descriptor de fichero que no estaba abierto devolverá error

Tabla 13. RF-11

ID	RF-12
Título	Función truncate
Descripción	La función debe de poder cambiar el tamaño de un archivo.
Tipo	Funcional
Importancia	Media

Pruebas de verificación	- Asegurarse de que el tamaño del archivo se cambia cuando se le aplica la función truncate - Si el archivo al que se le intenta aplicar la función truncate no existe, se devolverá error - Si se intenta aplicar la función a un directorio deberá de devolver error

Tabla 14. RF-12

ID	RF-13
Título	Función rename
Descripción	La función debe de poder cambiar el nombre de un archivo.
Tipo	Funcional
Importancia	Media
Pruebas de verificación	- Asegurarse de que si el archivo existía se cambia su nombre con éxito - Asegurarse de que, si se intenta aplicar a un directorio, la función devolverá error

Tabla 15. RF-13

ID	RF-14
Título	Función mkdir
Descripción	La función debe de poder crear un directorio, además de ser susceptible a crear una ruta de directorios si se le incluye un flag -p antes de la ruta que se desea crear, lo que creará una sucesión de directorios.
Tipo	Funcional
Importancia	Alta

Pruebas de verificación	- Asegurarse de que, si el directorio no existía, se crea - Asegurarse de que si el directorio ya existía la función devolverá error - Asegurarse de que la creación de una ruta de directorios mediante el flag -p crea con éxito los directorios que no existían y devuelve error en los que existían

Tabla 16. RF-14

ID	RF-15
Título	Función rmdir
Descripción	La función debe de poder eliminar un directorio o una ruta de directorios si se incluye un flag -p antes de la ruta que se desea borrar.
Tipo	Funcional
Importancia	Alta
Pruebas de verificación	- Asegurarse de que, si el directorio existía, se elimina con éxito - Asegurarse de que, si el directorio no existía, se devolverá un error - Asegurarse de que el borrado de una ruta de directorios mediante el flag -p borra con éxito los directorios que existían y devuelve error en los que no existían

Tabla 17. RF-15

ID	RF-16
Título	Función getcwd
Descripción	La función debe devolver la ruta de trabajo absoluta en la que se encuentra actualmente el usuario.
Tipo	Funcional
Importancia	Baja
Pruebas de verificación	Asegurarse de que se devuelve con éxito la ruta de trabajo actual.

Tabla 18. RF-16

ID	RF-17
Título	Estructura de directorio
Descripción	Se debe de generar una estructura donde se mantengan en memoria los directorios abiertos, así como los siguientes datos de cada directorio: - Descriptor de directorio - Posición de lectura - Estructura String_vector de ZooKeeper que contiene la lista de hijos con sus respectivos nombres y el número total de hijos.
Tipo	Funcional
Importancia	Alta
Pruebas de verificación	Asegurarse de que cuando se abre un directorio se rellena una de las estructuras de directorio.

Tabla 19. RF-17

ID	RF-18
Título	Función opendir
Descripción	La función debe devolver la estructura de directorio correspondiente a la ruta abierta.
Tipo	Funcional
Importancia	Alta
Pruebas de verificación	- Asegurarse de que se devuelve con éxito la estructura si el directorio existía - Asegurarse de que si el directorio no existía se devuelve error - Asegurarse de que no se puede abrir ficheros con esta función, si se intentase devolvería error

Tabla 20. RF-18

ID	RF-19
Título	Función readdir

Descripción	La función debe leer una estructura de directorio y devolverá estructuras de entrada en el directorio, correspondientes a uno de los hijos, la estructura de entrada tiene dos atributos: - Ruta de la entrada - Tipo de la entrada: DT_REG para archivos y DT_DIR para directorios
Tipo	Funcional
Importancia	Alta
Pruebas de verificación	- Asegurarse de que se devuelve con éxito la estructura si existen entradas restantes por leer - Asegurarse de que, si no quedan entradas restantes por leer o no había desde un principio, se devuelve NULL. - Asegurarse de que, si el directorio no está abierto, se devuelve error

Tabla 21. RF-19

ID	RF-20
Título	Función readdir_r
Descripción	La función debe llamar a la función definida en el requisito RF-19 de forma recursiva hasta leer todas las entradas del directorio abierto indicado.
Tipo	Funcional
Importancia	Media
Pruebas de verificación	Asegurarse de que se devuelven con éxito las entradas del directorio

Tabla 22. RF-20

ID	RF-21
Título	Función closedir
Descripción	La función debe de cerrar un directorio que esté previamente abierto
Tipo	Funcional
Importancia	Alta

Pruebas de verificación	- Asegurarse de que se cierra con éxito un directorio previamente abierto - Asegurarse de que se devuelve error al intentar cerrar un directorio que no haya sido abierto

Tabla 23. RF-21

ID	RF-22
Título	Función ls
Descripción	La función debe de mostrar los ficheros y directorios en la ruta actual, o en la indicada por parámetro
Tipo	Funcional
Importancia	Alta
Pruebas de verificación	Asegurarse de que si la ruta es válida se listan los directorios y archivos

Tabla 24. RF-22

ID	RF-23
Título	Función cd
Descripción	La función debe de entrar en un directorio, modificando así el directorio de trabajo actual
Tipo	Funcional
Importancia	Alta
Pruebas de verificación	Asegurarse de que si la ruta es válida se modifica el directorio de trabajo actual

Tabla 25. RF-23

ID	RF-24
Título	Función chmod
Descripción	La función debe modificar los permisos de un directorio o un fichero
Tipo	Funcional
Importancia	Alta

Pruebas de verificación	Asegurarse de que si la ruta es válida se modifican los permisos del directorio o fichero correspondiente indicado en la ruta

Tabla 26. RF-24

ID	**RF-25**
Título	Función stat
Descripción	La función debe de devolver los metadatos asociados al directorio o fichero indicado en la ruta.
Tipo	Funcional
Importancia	Alta
Pruebas de verificación	- Asegurarse de que si la ruta es válida se imprimen por pantalla los metadatos correspondientes al directorio o archivo - Asegurarse que los metadatos de fichero son distintos de los de directorio indicado en RF-06 y RF-07 - Si un directorio no tiene metadato se devolverá por pantalla que no existen metadatos asociados a la ruta indicada

Tabla 27. RF-25

ID	**RF-26**
Título	Función utime
Descripción	La función debe de modificar los metadatos de un directorio o fichero, actualizando su último tiempo de modificación al momento temporal en el que se invoque a la función.
Tipo	Funcional
Importancia	Alta
Pruebas de verificación	- Asegurarse de que si la ruta es válida se modifica el tiempo de última modificación - Asegurarse de que si no es un directorio o fichero con metadatos se devuelve error

Tabla 28. RF-26

ID	RF-27
Título	Función access
Descripción	La función debe de comprobar si se tiene el acceso indicado por parámetro a la ruta indicada, las posibles comprobaciones para el acceso son: - R_OK: Comprobar si se puede leer - W_OK: Comprobar si se puede escribir - X_OK: Comprobar si se puede ejecutar - F_OK: Comprobación de todos los permisos anteriores
Tipo	Funcional
Importancia	Alta
Pruebas de verificación	Asegurarse de que si la ruta es válida se devuelve correctamente si el proceso que llama a la función access tiene o no el permiso indicado por parámetro.

Tabla 29. RF-27

ID	RF-27
Título	Función read
Descripción	La función debe de un descriptor de fichero obtenido de la función open (RF-09) leer un número de bytes indicado en el read de los servidores de datos y eso se debe de devolver en un buffer que se pasa por parámetro.
Tipo	Funcional
Importancia	Alta
Pruebas de verificación	- Asegurarse de que si el descriptor de fichero es válido se hace una lectura - Asegurarse de que se devuelve el número de bytes leídos - Asegurarse de que se devuelve el mismo texto que hay contenido en los servidores de datos

Tabla 30. RF-27

ID	RF-28
Título	Función write
Descripción	La función debe escribir a un descriptor de fichero obtenido de la función open (RF-09) y escribir un número de bytes indicado en el write a los servidores de datos.
Tipo	Funcional
Importancia	Alta
Pruebas de verificación	- Asegurarse de que si el descriptor de fichero es válido se hace una escritura - Asegurarse de que se devuelve el número de bytes escritos - Asegurarse de que se realiza la escritura en los servidores de datos

Tabla 31. RF-28

ID	RF-29
Título	Función lseek
Descripción	La función debe de un descriptor de fichero obtenido de la función open (RF-09) leer un número de bytes indicado en el read de los servidores de datos y eso se debe de devolver en un buffer que se pasa por parámetro.
Tipo	Funcional
Importancia	Alta
Pruebas de verificación	Asegurarse de que si el descriptor de fichero es válido se hace un desplazamiento del puntero de lectura/escritura

Tabla 32. RF-29

ID	RF-30
Título	Servidor de datos
Descripción	El servidor de datos debe de tener dos funciones elementales: - PUT_BLOCK: Almacenar un bloque de datos en un fichero - GET_BLOCK: Leer un bloque de datos de un fichero
Tipo	Funcional

Importancia	Alta
Pruebas de verificación	- Asegurarse de que en modo PUT_BLOCK se almacena el bloque correctamente - Asegurarse de que en GET_BLOCK se devuelve el bloque leído correctamente

Tabla 33. RF-30

3.3.2. Requisitos no funcionales

ID	RNF-01
Título	Estándar POSIX
Descripción	Todas las funciones deben de tener el formato de parámetros, tipo de devolución y funcionalidad que se indica en el estándar POSIX de C.
Tipo	No Funcional
Importancia	Alta
Pruebas de verificación	Comprobar cada función para ver si cumple el estándar

Tabla 34. RNF-01

ID	RNF-02
Título	El lenguaje de programación deberá de ser C
Descripción	Todo el código debe de estar escrito en C.
Tipo	No Funcional
Importancia	Alta
Pruebas de verificación	Comprobar la extensión de los archivos

Tabla 35. RNF-02

ID	RNF-03
Título	Utilización de ZooKeeper
Descripción	El servidor de metadatos debe de estar implementado sobre ZooKeeper

Tipo	No Funcional
Importancia	Alta
Pruebas de verificación	El servicio de metadatos se deberá de levantar sobre ZooKeeper

Tabla 36. RNF-03

4. DISEÑO

En esta sección se expondrán las herramientas utilizadas para llevar a cabo el diseño del sistema para garantizar su funcionamiento final en base a los objetivos planteados en la introducción, así como el diagrama que describe el flujo de interacción del sistema para una mejor comprensión de dicho.

4.1. Elección de herramientas

Con motivo de la comparación de herramientas expuesta en el punto 3.1, cabe destacar que la utilización de la herramienta de ZooKeeper es esencial para el desarrollo de un sistema genérico de metadatos de forma que tenga una interfaz POSIX.

Respecto del entorno donde se lleva a cabo el desarrollo y las pruebas, se ha montado un entorno de máquina virtual que utiliza un sistema operativo basando en Ubuntu que es una conocida distribución de Linux. Al comienzo del desarrollo se utilizó Ubuntu, pero debido a una serie de errores del sistema operativo en mitad del desarrollo del proyecto se cambió a Elementary OS (que también está basada en Ubuntu). Cabe destacar que el desarrollo de este proyecto se pudo haber hecho en cualquier entorno siempre que fuese una distribución de Linux o perteneciente a Unix, con la ventaja de que son sistemas operativos totalmente gratuitos y de código abierto.

Para la programación no se ha utilizado ningún IDE (entorno de desarrollo integrado) como podría haber sido Visual Studio o Eclipse, sino que se ha desarrollado sobre un editor de texto con tabulación y marcado de código. Es un software de descarga y evaluación gratuita que facilita la programación ligeramente. Se podría haber usado cualquier otra herramienta que permitiese la edición de texto o código fuente, se eligió esta por uso previo y comodidad.

4.2. Diagrama de funcionamiento del sistema

A continuación, se presenta un diagrama con los elementos que se han desarrollado en el sistema y cuál es la interacción que tienen entre ellos. Los elementos que se representan son:

- Aplicación de usuario: La interfaz por la que se conecta un usuario y puede operar con el servicio de metadatos y con el de datos a través de la API. Este elemento también se conoce como el cliente (que accede a uno o varios servidores de forma transparente a través de la API).

- API desarrollada a partir de ZooKeeper: Son métodos en estándar POSIX donde muchos de ellos utilizan funciones existentes de ZooKeeper en su interior con configuración adicional para garantizar el correcto funcionamiento en el estándar POSIX. Mediante las llamadas a las funciones de ZooKeeper se produce la interacción con los servidores de ZooKeeper, mientras que en las operaciones de read y write se interacciona con los servidores de datos.
- Servidor/es de ZooKeeper: Son los que gestionan los metadatos de los clientes que consisten en información sobre cuál es el propietario del directorio o fichero, qué permisos existen para el acceso, la fecha de creación y última modificación entre otros.
- Servidor/es de datos: Desarrollado como prueba de concepto, para demostrar que el sistema de metadatos puede funcionar con uno de datos asociado, aunque aquí valdría cualquier servidor de datos. Gestionan la información relativa a los datos correspondientes a los ficheros del sistema de metadatos.

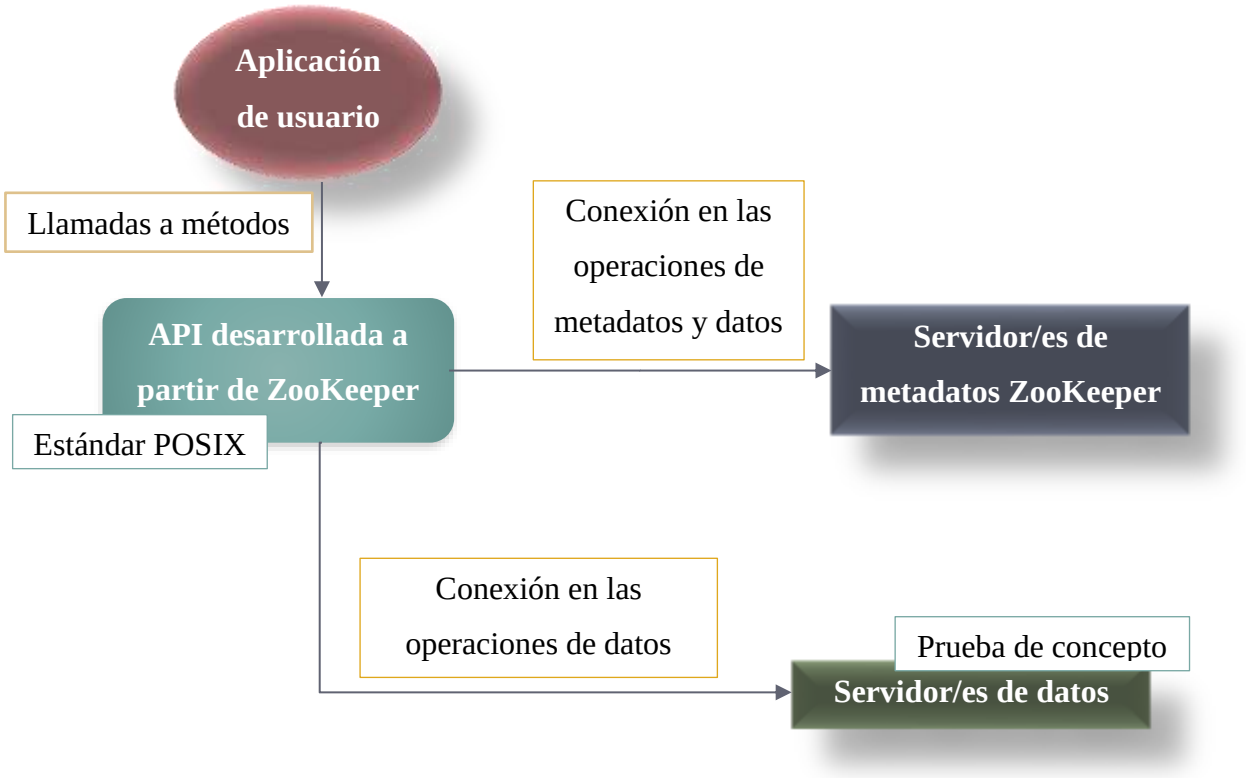


Ilustración 4-1. Diagrama de funcionamiento del sistema global

Cabe destacar que en esta arquitectura el diseño está enfocado en que sea modular y genérico, de forma que si se desea conectar el cliente a un sistema existente bastaría con ver cuáles son las características de ese sistema y adaptar ligeramente las funciones. Si en

vez del servidor de datos que se ha desarrollado como prueba de concepto se desea utilizar un sistema de almacenamiento de datos más conocido, también es posible y lo único que habría que cambiar es ligeramente el protocolo que tienen los servidores de metadatos de conectarse con los servidores de datos.

A continuación, se expondrán las estructuras de los metadatos, del sistema de ficheros completo, del sistema de datos y cómo accede un usuario al sistema de datos. Estos apartados ayudan a entender la arquitectura del sistema y la comunicación entre los componentes expuestos en la Ilustración 4-1.

4.3. Estructura de los metadatos

El objetivo de este apartado es explicar qué información es la que se almacena en el sistema de metadatos por cada fichero o directorio existente en el sistema. Esto representa la relación entre la API desarrollada, que es donde se define el contenido que se debe almacenar por cada fichero, y los servidores de metadatos de ZooKeeper, que es donde se almacena la información.

Para el diseño de estas estructuras, se comenzó a pensar en sistemas de ficheros tradicionales como los que se manejan a diario en los sistemas operativos de los ordenadores personales. Teniendo en mente que puede haber varios usuarios que utilicen el sistema de ficheros, se debe de establecer un propietario a cada uno de los ficheros creados en el sistema de ficheros.

Junto con el propietario, es conveniente tener una fecha de creación y de última modificación como información que puede ser útil a los usuarios. El tipo es importante distinguirlo también en un sistema de metadatos como ZooKeeper, ya que éste no distingue entre ficheros y directorios. En ZooKeeper todos los znodos son ficheros que pueden contener otros ficheros, en otras palabras, son directorios que pueden almacenar datos. El estándar POSIX no funciona acorde a esto, por ello se establece un parámetro “tipo” que indica si el znodo es un fichero o un directorio. Esto se define en el proceso de creación, en función de si el znodo se creó desde la función mkdir o creat/open.

Para garantizar la privacidad y seguridad de los archivos, es necesario almacenar en los metadatos los permisos de acceso a dicho fichero. El tamaño de un fichero o directorio también es importante; el tamaño de un fichero viene dado por la cantidad de información

que almacene, mientras que un directorio tiene tanto tamaño como todos los ficheros de sus interior.

La última información que comparten tanto ficheros como directorios es la ruta completa, incluyendo el nombre del fichero o directorio. Esto es útil para situar el znodo en el sistema de metadatos. Los sistemas de ficheros presentes en los sistemas operativos de los ordenadores personales también muestran esta información.

Por último, existen dos unidades de información que se almacenan exclusivamente en los metadatos de los znodos de tipo fichero.

En primer lugar, se trata de la información de dónde está contenido el primer bloque de datos, es decir, el nombre del servidor donde se encuentra (más adelante se verá por qué esto es importante).

En segundo lugar, el nombre original del fichero. Esto es necesario es necesario debido a la limitación de ZooKeeper de no permitir el renombramiento de nodos, la implementación que se ha realizado para renombrar nodos hoja (Fichero 1,2 y 3 en la ilustración inferior) consiste en el borrado y la creación del nodo con el nuevo nombre. El problema está en cuando ese nodo ya tenía datos almacenados en los servidores de datos, éstos identifican los datos mediante el nombre del fichero. Si éste se renombra, se deberían de renombrar todos los bloques asociados a él en los servidores de datos, pero esto es un procedimiento demasiado extenso. Por tanto, la solución es conservar el nombre original del fichero y gestionar los datos en base a ese nombre siempre.

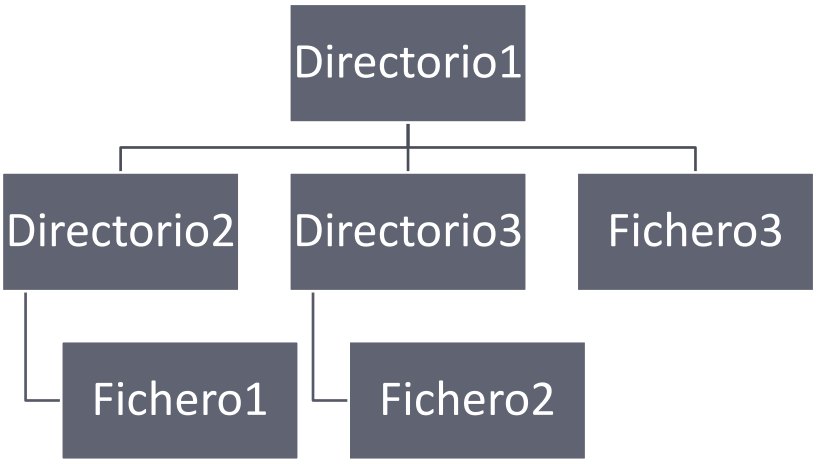


Ilustración 4-2. Ejemplo de jerarquía de directorios

A continuación, se muestran dos ilustraciones que representan los metadatos obtenidos con el comando de la API zstat respecto de nodos reales, concretamente de un directorio Dir1 que contiene dentro un fichero Fich1.

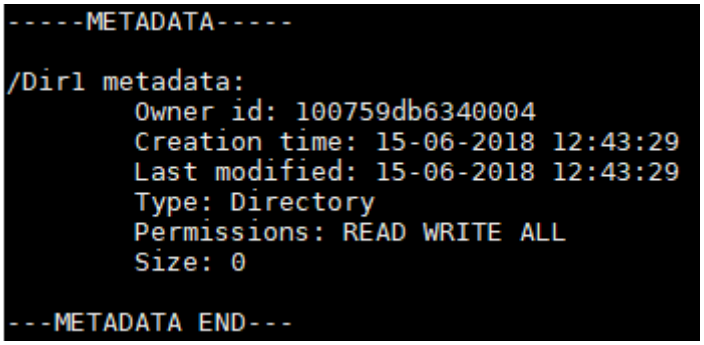


Ilustración 4-3. Metadatos de un directorio

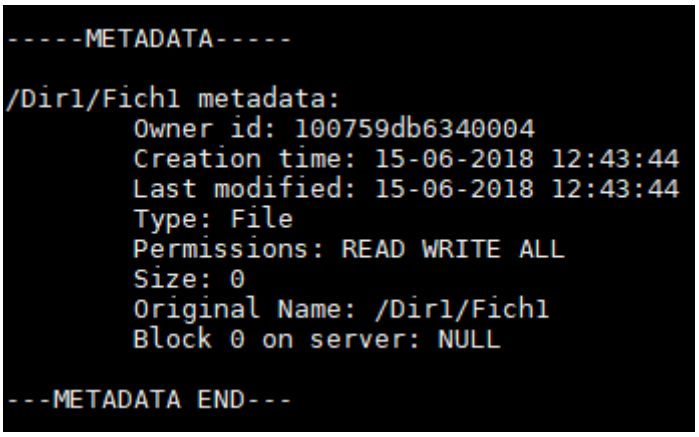


Ilustración 4-4. Metadatos de un fichero

4.4. Estructura del sistema de ficheros completo desde el punto de vista del servidor de datos

El diseño del servidor de datos y su correspondiente unión al servidor de metadatos fueron tareas complicadas desde el punto de vista de la sincronización de información entre ambos y establecer un protocolo definido. A continuación, se explicarán las dos cuestiones de diseño clave que explican la comunicación entre ambas partes.

Respecto de la implementación del servidor de datos, se divide en atender dos tipos de peticiones (tal y como se indica en el requisito RF-30):

- PUT_BLOCK: Cuyo protocolo es:
    - Esperar a obtener el mensaje que contiene la ruta del fichero.

    - Esperar a obtener el mensaje que contiene el tamaño de bloque de datos que se va a escribir.
    - Esperar a obtener el mensaje que contiene el bloque de datos.
    - Esperar a obtener el mensaje que contiene la posición del puntero dentro del bloque de datos.
    - Escribir el bloque de datos en un fichero con la ruta obtenida en el paso 1.
    - Devolver el número de bytes que se han escrito.
- GET_BLOCK: Cuyo protocolo es:
    - Esperar a obtener el mensaje que contiene la ruta del fichero.
    - Esperar a obtener el mensaje que contiene el tamaño de bloque de datos que se va a leer.
    - Esperar a obtener el mensaje que contiene la posición del puntero dentro del bloque de datos.
    - Leer el bloque de datos correspondiente.
    - Devolver el bloque de datos leído.

Cabe destacar que el protocolo de la librería correspondiente al servidor de metadatos es complementario, que significa que todo por lo que espera el servidor de datos, se lo tendrá que enviar el servidor de metadatos en ese orden.

Respecto del segundo detalle de diseño, consiste en la explicación general de qué metodología siguen las funciones de lectura y escritura desde la librería del servidor de metadatos, así como el uso que hace del servidor de datos.

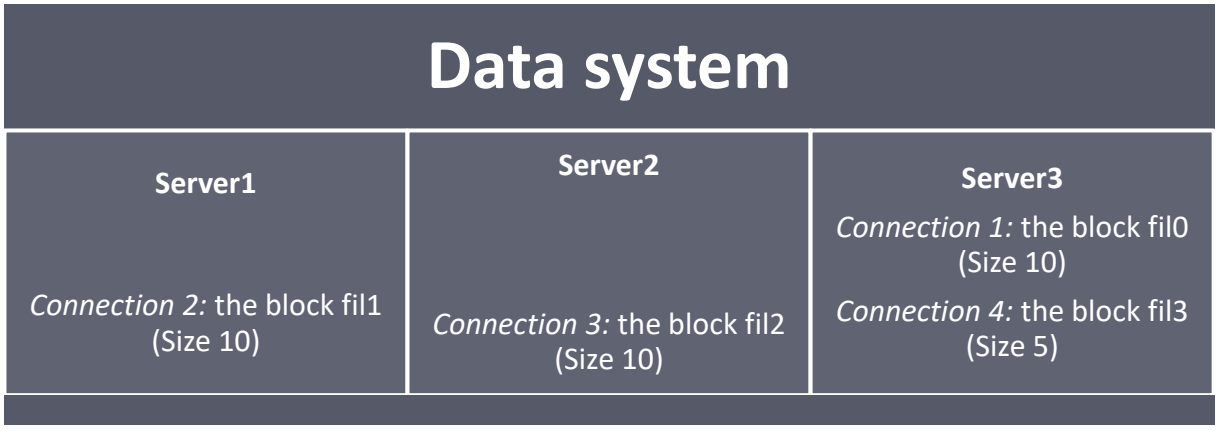
- Función de escritura zwrite: Se debe de hacer la comprobación de si el fichero ya existía en los servidores de datos, para ello se busca en los metadatos del fichero si existe un servidor donde se haya almacenado su bloque inicial. En caso de que no exista ese servidor, se tendrá que asignar uno. La forma en la que se asigna es haciendo una función resumen sobre la ruta del fichero y haciendo módulo sobre el número de servidores, de ahí se obtiene el número de servidor sobre el total al que se asigna el bloque inicial. Una vez existe un servidor que contiene (o contendrá) el bloque inicial se procede a mandar al servidor de datos el contenido que se indicó por parámetro de la función. En función del número de bloques en los que se subdivide el contenido a enviar, se harán tantas conexiones al servidor de datos como subdivisiones se hayan hecho.

- Función de lectura zread: El concepto más importante de la lectura es que, al igual que en la escritura, el número de llamadas al servidor de datos depende del tamaño de bloque y el tamaño de la lectura que se quiera hacer. El tamaño de la lectura se dividirá en bloques de, como máximo, el tamaño de bloque. Concatenando todos los resultados obtenidos en las distintas llamadas es como se obtendrá el resultado final, de la longitud indicada por parámetro o, si ésta era mayor que el tamaño del fichero, se leerá el tamaño del fichero.

4.5. Acceso del cliente a los datos

Desde el punto de vista del cliente, el sistema es sencillo, el cliente indica el fichero al que quiere escribir e indica una cadena de caracteres que desea almacenar en el fichero. El resto de las operaciones transcurren de forma transparente al usuario, pero lo que ocurre en realidad se muestra en el siguiente ejemplo.

- La aplicación de cliente se conecta a la API.
- Situación y configuración desde la API:
    - La operación es una escritura de un fichero llamado “Fichero” que se encuentra en una partición que tiene un tamaño de bloque 10 y existen 3 servidores de datos. Se desean escribir 35 caracteres.
    - Como es un fichero nuevo, se debe de asignar un servidor inicial a partir de un algoritmo de resumen; la suma de los valores ASCII del nombre de fichero es 704, haciendo módulo sobre 3 obtenemos que el primer bloque debe de colocarse sobre el tercer servidor.
- Organización en los servidores de datos:



Como podemos observar, al nombre original del fichero “Fichero” se le pone una extensión en forma de números naturales comenzando en el 0. El servidor inicial viene dado por el algoritmo de resumen (hash) visto anteriormente. Los servidores sucesivos se eligen mediante el algoritmo Round Robin, que no es más que elegir el servidor de la derecha, y si es el último volver al primero. El fichero se divide en tantas partes como haga falta dado el tamaño de bloque, en este caso fueron 4 partes debido a que el tamaño de bloque es 10 y el tamaño del fichero es 35, resultando en 3 bloques entero y uno a la mitad.

Respecto a la lectura, en base al número de servidores, la cantidad de bytes que se desean leer y el tamaño de bloque, se realiza un cálculo para determinar el número de conexiones (o bloques distintos) que hay que leer. El retorno de la lectura es la concatenación de todos esos bloques.

5. IMPLEMENTACIÓN E IMPLANTACIÓN

En este apartado se expondrá en la parte de implementación los detalles más complicados que se han experimentado en la realización del sistema de ficheros. En el apartado de implantación se expondrá brevemente el procedimiento para instalación del sistema de ficheros.

5.1. Implementación

El primer problema que surgió fue el cómo empezar a desarrollar una librería para el sistema de ficheros, desde la primera función que fue la de crear un directorio. El procedimiento que se siguió para entender la interacción de ZooKeeper entre servidores y clientes fue a base de ejecutar un servidor y el cliente base que viene con ZooKeeper y probar a realizar las funciones básicas y después comprobar cómo están implementadas en código. En la implementación realizada en nuestro sistema de ficheros, se utiliza una estructura similar en la aplicación de usuario respecto del flujo de ejecución del programa.

Una vez resuelto el primer estancamiento, se podía proceder a implementar la primera función, que corresponde a la creación de directorios, cuyo principal problema fue el manejo de permisos, ya que se tenía en mente utilizar unos permisos en estándar POSIX, los mismos que se utilizan en sistemas operativos Linux, que están basados en un sistema octal y separado en permisos de usuario, grupo al que pertenece al usuario y resto de usuarios. La forma en la que se ha decidido traducir al final se expone a continuación:

- Se comprueba el modo pasado por parámetro y se compara con los permisos de lectura, escritura y ejecución de usuario en el sistema octal de Linux. En caso de coincidir alguno de los permisos del modo pasado por parámetro con los del sistema octal, se adjunta a la lista de accesos de ZooKeeper el permiso correspondiente. El permiso de ejecución se traduce a todos los permisos en ZooKeeper, mientras que el de lectura y escritura se traducen a lectura y escritura en ZooKeeper. Para completar la explicación creo que hace falta adjuntar el fragmento de código en concreto:

```
//Inicialización de la variable MYACL que representará los permisos
struct ACL MYACL[] = {{0, ZOO_AUTH_IDS},{0, ZOO_ANYONE_ID_UNSAFE}};


//Comparación del modo pasado por parámetro con los permisos octales de linux
if(mode & S_IRUSR){
        //En caso de que coincida, se añade el permiso correspondiente a la ACL
        MYACL[0].perms |= ZOO_PERM_READ;
}
if(mode & S_IWUSR){
        MYACL[0].perms |= ZOO_PERM_WRITE;
}
if(mode & S_IXUSR){
        MYACL[0].perms |= ZOO_PERM_ADMIN | ZOO_PERM_CREATE |
ZOO_PERM_WRITE | ZOO_PERM_DELETE | ZOO_PERM_READ;
}
```

Ilustración 5-1. Código referente a la traducción de permisos de POSIX a ZooKeeper

El siguiente problema de implementación consiste en que ZooKeeper maneja las rutas de forma absoluta, lo que esto significa es que, si se quiere crear un directorio llamado d1 que tenga un fichero f1 y otro f2 en su interior, se tiene que invocar a la función de creación de directorio con la ruta d1 y la función de creación de fichero con la ruta d1/f1 y otra vez con d1/f2. Esto es sencillo con poca profundidad, pero en el supuesto caso de querer crear dos ficheros en una profundidad de 10 directorios las rutas se vuelven tediosas de gestionar. Para solucionar este problema se ha implementado un sistema de rutas relativas de forma que se mantiene en una variable global en todo momento el directorio de trabajo actual (*current working directory o cwd*), y unas funciones de navegar dentro y fuera de directorios, así como de listar el contenido del directorio actual. Volviendo al ejemplo anterior, en este caso se podría crear el directorio d1, entrar dentro de él (cwd es d1) y crear un fichero f1 y otro f2, se crearán en las rutas d1/f1 y d2/f2 respectivamente de forma automática.

El siguiente punto es un conjunto de problemas que se solucionaron todos de la misma forma, en esencia el problema consiste en que cuando se abre un directorio o fichero con la librería estándar de POSIX se obtiene en el caso del directorio una estructura DIR, dentro de la cual cada lectura de DIR devuelve un struct dirent; mientras que la apertura de un fichero devuelve un descriptor de fichero o fd. La cuestión reside en que no es posible utilizar estas estructuras propias de POSIX y se debe implementar cada una manualmente y renombrarla para que no haya colisiones con la librería nativa de POSIX de sistemas de ficheros. La información que contiene cada una de estas estructuras está oculta por POSIX por defecto, pero en base a las necesidades respecto de las distintas funciones a implementar se ha llegado a los siguientes atributos dentro de las estructuras:

```
struct ZDIR {
        int fd; //Descriptor de fichero
        off_t filepos; //Siguiente posición de lectura
        struct String_vector * vector; //Almacena nombres de hijos y el número de hijos
        char path[PATH_LENGTH]; //Ruta del directorio abierto
};
```

Ilustración 5-2. Estructura DIR para la librería de ZooKeeper

```
struct ZFILE {
        int fd; //Descriptor del fichero
        off_t filepos; //Puntero de lectura/escritura
        char parent[PATH_LENGTH]; //Ruta del padre
        char path[PATH_LENGTH]; //Ruta del fichero abierto
        int perms; //0 si solo lectura 1 si solo escritura, 2 si lectura/escritura
        int size; //Tamaño del archivo
};
```

Ilustración 5-3. Estructura FILE para librería de ZooKeeper

En los siguientes epígrafes se describirán en detalle las partes de la aplicación de usuario, la API y el servidor de datos. Se explicarán con un par de funciones en cada una cuál es la implementación que se ha seguido con algunas muestras de código.

5.1.1. Aplicación de usuario

Esta aplicación se ha nombrado myclient y su código dentro del main consiste en un bucle infinito que acepta comandos del usuario. A continuación, se muestra la parte del código correspondiente:

```
while(!shutDown){
    printf("myclient> ");
    fflush(stdout);
    int rc;
    int len = sizeof(buffer) - bufoff - 1;
    rc = read(0, buffer+bufoff, len);
    bufoff += rc;
    buffer[bufoff] = '\0';
    while (strchr(buffer, '\n')) {
        char *ptr = strchr(buffer, '\n');
        *ptr = '\0';
        processline(buffer);
        ptr++;
        memmove(buffer, ptr, strlen(ptr)+1);
        bufoff = 0;
    }
}
```

Ilustración 5-4. Código main de myclient

Dentro del bucle, después de cada comando introducido se imprime de nuevo en la consola un "myclient>" para indicar al usuario que puede seguir introduciendo comandos. Después, la impresión por pantalla carece de un salto de línea en la línea siguiente hacemos un "flush" del buffer de salida para garantizar que myclient se imprime siempre.

El punto clave está dentro del segundo bucle while, que está anidado dentro del primero. En éste se llama a la función processline con la línea que ha introducido el usuario por comando y se evalúa si ésta corresponde a un comando existente.

Ya que la función processline contiene tantos if / else if como comandos existen en nuestro sistema de ficheros, se muestran solo algunos de los comandos implementados en el sistema de metadatos. Estos comandos de ejemplo sirven como representación de la lógica de diseño de toda la aplicación de cliente.

```
if (startsWith(line, "ls")) {
        char * ptr = strchr(line, ' ');
        char * newline = malloc(strlen(line)+1);
        if (ptr != NULL) {
            *ptr = '\0';
            ptr++;
            if (ptr[0] != '/') sprintf(newline, "/%s", ptr);
            else newline = ptr;
            rc = zls(newline);
        }else{
            rc = zls("/");
        }
        printf("CLIENT LS: RC = [%d]\n",rc);
}
```

Ilustración 5-5. Comando ls de myclient

En este condicional se observa que se comprueba que la línea pasada por parámetro desde el main comience con "ls", siendo este comando el que lista el contenido de un directorio. Este directorio puede ser pasado por parámetro o, si no se pasa ninguno, se listará el contenido del directorio de trabajo actual. Observamos que se busca que la línea empiece con "ls" que no contiene un espacio al final como veremos con los comandos siguientes justamente para soportar que no se indique ningún parámetro.

Como es costumbre, se realiza una impresión por pantalla después de que se haya hecho la llamada a la API (mediante "zls") recogiendo el retorno de la función, que indica si la función tuvo éxito (conocido en ZooKeeper como "ZOK" que tiene asociado el valor 0), o algún fallo (dado por un valor distinto de 0).

A continuación, se indica la rutina de ejecución cuando el comando introducido por el usuario se corresponde con "rmdir", que se corresponde con el comando que elimina un directorio.

```
if (startsWith(line, "rmdir ")) {
        line += 6;
        char * newline = malloc(strlen(line)+1);

        if (line[0] != '/') sprintf(newline, "/%s", line);
        else newline = line;

        rc = zrmdir(newline);
        printf("CLIENT RMDIR: RC = [%d]\n",rc);
}
```

Ilustración 5-6. Comando rmdir de myclient

Podemos observar respecto del comando anterior que éste busca en todos los casos un parámetro, que en este caso debe corresponderse al nombre (o ruta) de un directorio. Esta ruta se introduce por parámetro a la función de la API desarrollada zrmdir y como último paso, se captura el retorno para comprobar si el comando tuvo éxito u ocurrió un error.

Ahora que se ha expuesto la arquitectura de este tipo de comandos que se corresponden más bien con operaciones de metadatos, se procederá a contrastar esto con el diseño de los comandos que actúan sobre datos; desde el punto de vista del cliente.

```
if(startsWith(line, "write ")){
        line += 6;

        char * ptr = strchr(line, ' ');

        int rc;
        if(ptr != NULL){
            *ptr = '\0';
            ptr++;

            char * fd = malloc (strlen(line)+1);
            strcpy(fd,line);

            rc = zwrite(atoi(fd), ptr, strlen(ptr));
        }else{
            printf("Wrong write arguments\n");
            rc = -2;
        }
        printf("CLIENT WRITE: RC = [%d]\n",rc);
    }
```

Ilustración 5-7. Comando write de myclient

En este caso, existe más de un parámetro que introducir. El primero se corresponde con el número del descriptor de fichero al que queremos escribir. Se obtiene un descriptor de fichero después de realizar la función “open” sobre un fichero. Se pueden tener hasta 16 ficheros abiertos al mismo tiempo, por ello parece lógico que el cliente pueda elegir en cuál de esos posibles 16 ficheros abiertos puede escribir.

El segundo parámetro de la función write es la cadena de caracteres, buffer o, en otras palabras, el mensaje que se quiere escribir en el fichero que está asociado al descriptor de fichero. Si no existen dos parámetros la función falla con el número de error -2, en caso de que el número de parámetros sea correcto, el valor de retorno será dado por la función de la API desarrollada zwrite.

Vistos los dos tipos de comandos que existen en la aplicación de cliente, unos de metadatos y otros de datos, quedan dos comandos especiales que sirven para configurar el sistema de ficheros en el que se manejará el usuario.

El primero sirve para hacer una partición con un nombre que se pasa por el primer parámetro y un tamaño de bloque se pasará por el segundo parámetro, el código se ilustra en la siguiente imagen.

```
if(startsWith(line, "makepart ")){
        line += 9;

        char * newline = malloc(strlen(line)+1);
        int rc;
        char * ptr = strchr(line,' ');
        if(ptr != NULL){
            *ptr = '\0';
            ptr++;
            strcpy(newline, line);
            rc = make_part(newline, atoi(ptr));
        }else{
            rc = -1;
        }
        printf("CLIENT MAKEPART: RC = [%d]\n",rc);
}
```

Ilustración 5-8. Comando makepart de cliente

La función de la API a la que se llama es make_part, que es la que realiza el trabajo para montar la partición en la jerarquía correspondiente:

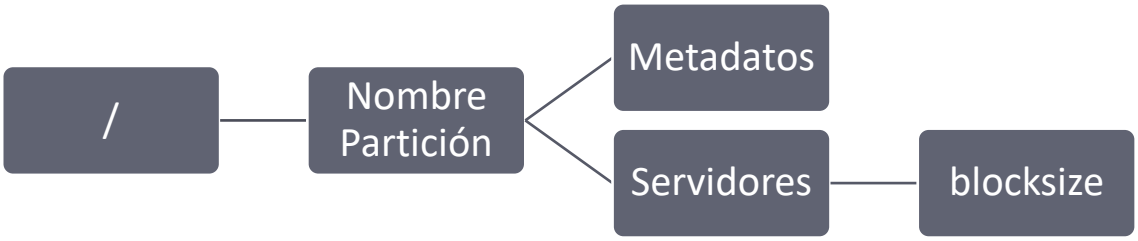


Ilustración 5-9. Jerarquía cuando se crea una partición

En el diagrama previo existen cinco directorios cuya explicación es:

- /: Directorio raíz, este es el directorio padre creado por defecto a partir del cual se crean particiones.
- Nombre Partición: Es el nombre introducido en el primer parámetro de la función make_part, contiene dos subdirectorios y nada más.
- Metadatos: Contiene todos los directorios y ficheros que el usuario desee crear.
- Servidores: Contiene la información de los servidores de datos que existan. La información que tiene cada servidor es IP y puerto.
- Blocksize: Es el tamaño de bloque del sistema de datos que se utilizará en toda la partición por los servidores de datos. El tamaño de bloque no es más que cuánto ocupa cada "trozo" (o bloque) de información en los servidores de datos. Sirve para estructurar la información de forma ordenada y constante.

Para la creación de servidores, se ejecutará el comando add_server con cuatro parámetros, el primero indica el nombre de la partición en el que se añadirá, el segundo será el nombre simbólico del servidor, que no tiene ningún significado más allá de que el cliente le otorgue un nombre. Por último, los últimos dos parámetros son la IP y el puerto donde se encontrará el servidor de datos. El servidor se añadirá bajo la ruta /nombrePartición/servidores/nombreServidor/. El código del cliente es el siguiente:

```
if(startsWith(line, "addserver ")){
        line += 10;
        char * newline = malloc(strlen(line)+1);
        char * ip = malloc (32);
        int rc;
        char * ptr = strchr(line,' ');
        if(ptr != NULL){
            *ptr = '\0';
            ptr++;
            strcpy(newline, line);
            char * ipptr = strchr(ptr,' ');
            if(ip != NULL){
                *ipptr = '\0';
                ipptr++;
                strcpy(ip,ptr);
                char * puerto = strchr(ipptr,' ');
                if(puerto != NULL){
                    *puerto = '\0';
                    puerto++;
                    rc = add_server(newline, ptr, ip, puerto);
        }
        printf("CLIENT MAKEPART: RC = [%d]\n",rc);
}
```

Ilustración 5-10. Comando addserver de myclient

5.1.2. API desarrollada a partir de ZooKeeper

Al igual que en el apartado anterior, se van a explicar algunas de las funciones para dar una idea general de cómo funciona en su totalidad, en la API hay funciones que ocupan bastante y no es práctico poner todo el código así que las funciones que sean más extensas se explicarán por trozos. Aun así, no se pondrán trozos de código que sean poco importantes respecto de la funcionalidad.

Aprovechando que se explicó la función ls en el apartado anterior, y que es una función relativamente corta y sencilla, además que hace una llamada a la API de ZooKeeper, es un método ideal para entender el funcionamiento básico de la interfaz desarrollada.

```
int zls(char* path){
    char * ret = connection_init();
    if(ret == NULL) return -1;

    int rc;

    char tempPath[PATH_LENGTH];

    if(strlen(currdir)>1)strcpy(tempPath,currdir);
    else strcpy(tempPath,"");
    if(strlen(path)>1) strcat(tempPath,path);

    if(endsWithSlash(tempPath) == 0) tempPath[strlen(tempPath)-1] = '\0';
    if(strlen(tempPath) == 0) strcpy(tempPath,"/");

    rc = zoo_aget_children(zh, tempPath, 0, my_strings_completion, NULL);

    return rc;
}
```

Ilustración 5-11. Función zls de la API desarrollada

El procedimiento seguido en esta función es similar al resto y consiste en unos pasos definidos:

- connection_init(): Realiza la conexión con el servidor de ZooKeeper
- Manipulación del parámetro path para adaptarlo al directorio de trabajo actual en el que se encuentra al usuario, para dar soporte a rutas relativas (desde el directorio actual)
- Llamada a la función de la API de ZooKeeper que realiza la misma función que se necesita, o una función similar con la que se puede trabajar para terminar consiguiendo la funcionalidad deseada. En este caso zoo_aget_children realiza exactamente lo que queremos, lista el contenido de un directorio.

La siguiente función que se explicará, no es ninguna relacionada con el estándar POSIX, es decir no es una operación directa que tenga una traducción directa sobre sistemas de ficheros convencionales, es más bien una función auxiliar que se ha utilizado en muchos de los métodos. Su propósito es modificar el valor de un parámetro de los metadatos y, además, devolver la estructura de metadatos modificada. También existe la opción de no modificar ningún parámetro y utilizar la función para devolver la estructura para consultarla tal y como está.

Esta función en realidad solo puede utilizarse para directorios, pero existe otra que es para ficheros que tiene la posibilidad de manipular todos los parámetros que puede manipular la de directorios y dos más, por esta razón se explicará la función que corresponde con la modificación de metadatos de ficheros.

```
znode_fmetadata_char changeFileStruct(char * path, int id, char * change){
    znode_fmetadata_char meta;
    char * ret = connection_init();
    if(ret == NULL) return meta;

    char data[8192];
    char backup[8192];
    int length = 8192;
    char * ptr;
    char * end;

    zoo_get(zh, path, 0, data, &length, NULL);
```

Ilustración 5-11. Función auxiliar changeFileStruct parte 1

Al comienzo se inicia una conexión con el servidor de ZooKeeper si no existía ya. Después se declaran una serie de variables y se llama a la función zoo_get que extrae los metadatos almacenados en la ruta pasada por parámetro path. Los metadatos extraídos quedan almacenados en la variable data, y su tamaño en bytes en la variable length.

En la siguiente ilustración se continúa con que, mientras que los metadatos que sea hayan extraído no sean nulos, en cada iteración se extrae el valor de cada uno de los parámetros para almacenarlos en la estructura declarada en la parte 1 al principio.

En caso de que el parámetro de id sea de 0 a 7, se cambiará el parámetro elegido por la id por el nuevo valor indicado por el parámetro change.

```
if (data != NULL && strstr(data,path) != NULL && length > 0) {
    int i;
    for(i = 0; i <=7; i++){
        strcpy(backup,data);
        switch(i){
             case 0: ptr = strstr(backup, CHR_OWNERID);
                     ptr+=strlen(CHR_OWNERID); break;
                  .
                  .
                  .

             case 7: ptr = strstr(backup, CHR_BLCKSV);
                     ptr+=strlen(CHR_BLCKSV);break;
    }

    end = strchr(ptr, '\n');
    *end = '\0';

    switch(i){

        case 0: meta.client_id = malloc(sizeof(ptr));
                strcpy(meta.client_id,ptr);
                if(id == i) strcpy(meta.client_id, change);break;
                  .
                  .
                  .
        case 7: meta.servidorInicial = malloc(sizeof(ptr));
                strcpy(meta.servidorInicial, ptr);
                if(id == i) strcpy(meta.servidorInicial, change); break;
    }
}
```

Ilustración 5-12. Función auxiliar changeFileStruct parte 2

Por último, si el parámetro que se desea cambiar es el path que aparece en la primera línea de los metadatos, que indica la ruta del fichero, se realiza el cambio. Seguidamente se hace la copia de la estructura de metadatos que se ha ido recopilando a lo largo de la función a la variable de datos para, en caso de que se haya realizado algún cambio (indicado por la variable id), persistirla en el sistema de metadatos.

```
    if(id == FMETAPATH) strcpy(meta.path, change);
    else strcpy(meta.path,path);

    strcpy(data, fchrStruct2str(meta));

    if (id != -1) zoo_set(zh, path, data, strlen(data), -1);

    return meta;
}
```

Ilustración 5-13. Función auxiliar changeFileStruct parte 3

La siguiente función que se va a explicar es una de las más completas y largas, es la función zwrite que se encarga de escribir en un fichero previamente abierto. Trabaja con metadatos y datos, también cumple un protocolo respecto del servidor de datos mediante conexiones basadas en sockets.

```
ssize_t zwrite(int fd, const void *buf, size_t count){
    char * ret = connection_init();
    if(ret == NULL) return -1;

    if(strcmp(fdfiles[fd]->path,"") == 0){
        return -1;
    }

    if(fdfiles[fd]->perms == 1 || fdfiles[fd]->perms == 2){

      znode_fmetadata_char meta = changeFileStruct(fdfiles[fd]->path, -1,
      NULL);

        char backup[PATH_LENGTH];
        strcpy(backup, meta.nombreoriginal);

        char * ptr = strchr(meta.nombreoriginal+1,'/');
        *ptr = '\0';
        ptr = meta.nombreoriginal;

        char temp[PATH_LENGTH];
        sprintf(temp, "%s/servidores", ptr);
        strcpy(meta.nombreoriginal, backup);
```

Ilustración 5-14. Función zwrite parte 1

Como toda función de la API, comienza con el intento de conexión a los servidores de ZooKeeper. Después se comprueba que el descriptor de fichero introducido está abierto. Es necesaria una comprobación de si existen permisos de escritura sobre ese descriptor de fichero, ya que es la operación que se realiza en esta función. Se consiguen los metadatos del fichero gracias a la función auxiliar descrita anteriormente. Se averigua si el fichero está en un sistema de partición y si es así se accede a los servidores de dicha partición.

```
struct String_vector * vector = malloc(sizeof(struct String_vector));
        zoo_get_children(zh, temp, 0, vector);
        int numSvers = vector->count - 1;

        if(numSvers == 0){
            return -1;
        }

        char temp2[PATH_LENGTH];
        sprintf(temp2, "%s/blocksize", temp);

        int blocksize = getPartitionBlocksize(temp2);
        if(blocksize == -1){
            return -1;
        }
```

Ilustración 5-15. Función zwrite parte 2

Se reserva memoria para la estructura string_vector que es utilizada por la llamada zoo_get_children para devolver los contenidos del directorio de servidores de la partición donde se encuentra el fichero al que se desea escribir. Al número de nodos hijos encontrados se le debe restar uno para quedarse con el número de servidores de la partición, ya que en el directorio de servidores también se encuentra el tamaño de bloque utilizado por la partición. En caso de que existan servidores, se leerá el tamaño de bloque.

```
int currentBlock = fdfiles[fd]->filepos / blocksize;
        int relativeFilepos = fdfiles[fd]->filepos % blocksize;

        int serversNeeded = 0;
        if((relativeFilepos + count) % blocksize == 0) serversNeeded =
(relativeFilepos + count) / blocksize;
        else serversNeeded = ((relativeFilepos + count) / blocksize) + 1;

        char *** fullIP = malloc(numSvers * sizeof(char**));
        for(int i = 0; i < numSvers; i++){
            fullIP[i] = malloc(2 * sizeof(char *));
            fullIP[i][0] = malloc(15 * sizeof(char));
            fullIP[i][1] = malloc(5 * sizeof(char));
        }
```

Ilustración 5-16. Función zwrite parte 3

Mediante el tamaño de bloque se calcula el bloque en el que se comenzará a escribir, así como en qué lugar de ese bloque se hará la escritura. Después se calculará el número de accesos a servidores que se necesitan. Con esta información es posible rellenar la matriz llamada fullIP que contendrá información de servidores de datos donde:

- En la primera dimensión se reserva espacio para dos campos que serán arrays
- La segunda dimensión, primer componente, se corresponde con la IP
- La segunda dimensión, segundo componente, se corresponde con el puerto

Después viene una parte del código que se explica con un ejemplo en el apartado 4.5, donde se especifica cuál es la elección de servidor de datos inicial en el caso de que no exista ninguno asignado (primera escritura en el fichero).

En la ilustración siguiente (parte 4), después de inicializar la variable que servirá de contador de cuánto se lleva escrito, así como de declarar un puntero que apunta al comienzo de la cadena de caracteres que se pasó por parámetro; se comienza con el bucle que realiza las conexiones a todos los servidores necesarios para persistir el mensaje pasado por parámetro.

```
int totalresp = 0;
char bufcopy[strlen(buf)];
strcpy(bufcopy,buf);
ptr = bufcopy;

for(int i = 0; i < serversNeeded; i++){

      int rc = connect_to_dataserver(fullIP[i%numSvers][0],
fullIP[i%numSvers][1]);
      if (rc != 0){
            return -1;
      }
```

Ilustración 5-17. Función zwrite parte 4

La primera operación realizada es la conexión al servidor de datos que toca en el momento. Si no se pudo completar la conexión con éxito se terminará el proceso de escritura.

El bloque de código que viene a continuación es muy largo y se resume en seguir el protocolo que se indica en el apartado 4.4, enviando línea a línea la información necesaria para cumplir el protocolo. Se va a avanzar hacia qué ocurre cuando se han terminado todas las escrituras en los servidores de datos.

El algoritmo desarrollado, en este punto debe de persistir las modificaciones realizadas con el fichero en los metadatos, modificando el tamaño del fichero. Para realizar la modificación del tamaño se han de tener en cuenta varios factores: el tamaño del fichero antes de la escritura, la posición del puntero y la cantidad de caracteres escritos.

Existen tres escenarios definidos que se entienden mejor con ejemplos:

- El tamaño del fichero es 10, el puntero de escritura es 0 y se escriben 5 caracteres en el fichero. Esto resulta en una sobreescritura de los 5 primeros caracteres del fichero, pero su tamaño sigue siendo el mismo, por tanto, lo único que se modifica es la posición del puntero de escritura a 5 (número de caracteres escritos).
- El tamaño de fichero es X, el puntero de escritura está en la misma posición X (al final del fichero). Por tanto, cualquier cantidad escrita Y aumenta el tamaño del fichero en una cantidad Y; el puntero de escritura también aumenta en la misma cantidad.
- El tamaño de fichero es 10, el puntero de escritura es 5 y la cantidad escrita son 10. Este escenario resulta en un aumento del tamaño de 5, resultando en un tamaño total del fichero de 15. Esto se debe a que se han sobrescrito los últimos 5

caracteres del fichero y se han añadido 5 nuevos. El puntero de escritura se posicionaría al final, en la posición indicada por el tamaño total del fichero (15).

Después de persistir los cambios en los metadatos, se procede a devolver el número de caracteres escritos como valor de retorno de la función. Este valor no siempre es el mismo que se indicó inicialmente por parámetro, ya que en realidad el máximo es el tamaño del buffer indicado por parámetro. Aunque la cantidad escrita sí que puede ser menor que el tamaño de buffer, nunca podrá ser mayor.

Con esto se concluye la explicación de la API ya que se cree que esto da una idea general de cuáles fueron los problemas más complicados a abordar en el desarrollo de la API y cuáles fueron las peculiaridades de la implementación.

5.2. Implantación

Para instalar el entorno hace falta descargarse una versión de ZooKeeper, aunque se ha trabajado sobre la versión 3.4.11 no tendría que haber ningún problema con versiones anteriores ni posteriores.

5.2.1. Compilación

Una vez descargado el archivo comprimido de ZooKeeper se puede descomprimir en el lugar deseado. Los archivos que se han desarrollado posix_fs_calls.h, posix_fs_calls.c y myclient.c se posicionarán en la ruta relativa respecto del directorio raíz de ZooKeeper /src/c/. La carpeta del servidor de datos se posicionará sobre /src/c/servidor/ cuyos archivos son read_line.h, read_line.c y servidor.c. En este mismo directorio se creará un directorio nombrado almacenamiento de forma automática cuando se hagan operaciones de escritura al servidor de datos.

Para compilar los programas de usuario y las librerías se han utilizado los siguientes comandos (en /src/c/):

- **Para compilar la API:**

  gcc -o posix_fs_calls.o -c posix_fs_calls.c

- **Para compilar el cliente y enlazarlo con la API:**

  sudo libtool --mode=link gcc -O -o myclient myclient.c servidor/read_line.c -DTHREADED posix_fs_calls.o ./libzookeeper_mt.la

Para compilar el servidor (src/c/servidor):

```
gcc -pthread servidor.c read_line.c -o servidor -Wall -Werror
```

5.2.2. Ejecución

Para inicializar todo el entorno después de haber compilado, en primer lugar, se inicializa el servidor de ZooKeeper mediante la ejecución del siguiente comando desde el directorio principal:

java -cp zookeeper-3.4.11.jar:lib/log4j-1.2.16.jar:lib/slf4j-log4j12-1.6.1.jar:lib/slf4j-api-1.6.1.jar:conf org.apache.zookeeper.server.quorum.QuorumPeerMain conf/zoo.cfg

Después, para inicializar el cliente que hace uso de la API se puede hacer mediante el siguiente comando: ./myclient ip:puerto

Y para inicializar el servidor de datos, se puede hacer con el siguiente comando: ./servidor ip:puerto

En ambos comandos el puerto es un parámetro que debe de estar entre 1024 y 65535. La IP es la correspondiente al servidor que desea hacer la conexión en el caso del cliente. Por otra parte, la IP del servidor de datos se corresponde con dónde se quiere publicar el servidor de datos para que éste atienda peticiones.

5.2.3. Librerías necesarias

Para la correcta compilación de los archivos desarrollados, especialmente para enlazar la API con la aplicación de usuario, hacen falta unas librerías adicionales de Linux a parte de las que vienen preinstaladas como gcc:

- ant
- autoconf
- libcppunit-dev
- g++
- libtool-bin

6. EVALUACIÓN

En este apartado se expondrán una serie de datos que muestran el rendimiento de nuestro sistema de metadatos y sistema de ficheros en diversas pruebas. Tanto el sistema de metadatos como el de datos se va a evaluar midiendo el tiempo de ejecución de distintas funciones de la API desarrollada y realizando una comparación respecto del sistema de ficheros propio de Linux.

Las pruebas respecto de la API desarrollada, relacionado con los servidores de metadatos, se realizarán sobre 1 único servidor de metadatos, 3 servidores de metadatos y 9 servidores de metadatos.

Las pruebas del sistema de datos se realizarán comparando el tiempo que tardan las operaciones de lectura y escritura utilizando un tamaño de 8kb y 256kb.

6.1. Evaluación del sistema de metadatos

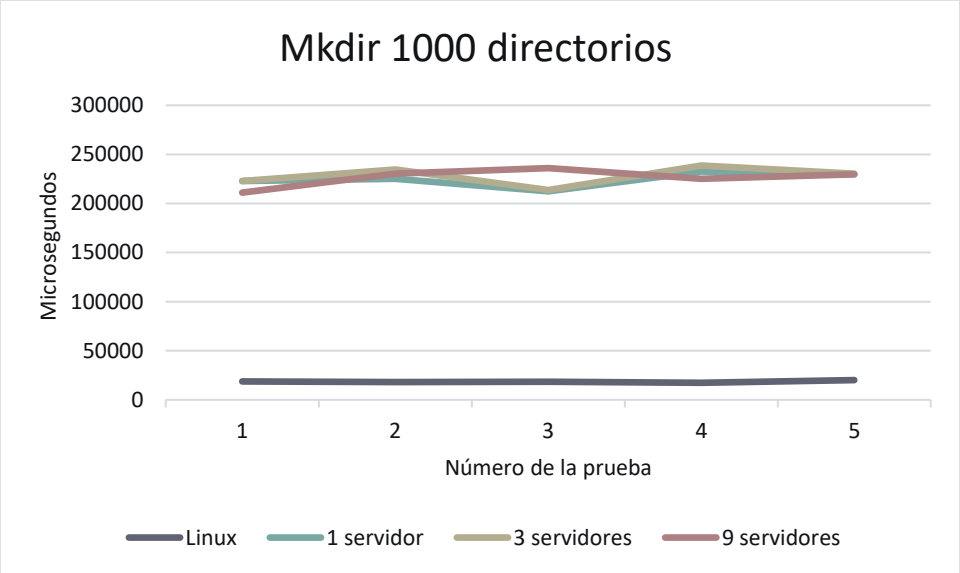


Ilustración 6-1. Prueba mkdir 1000 directorios

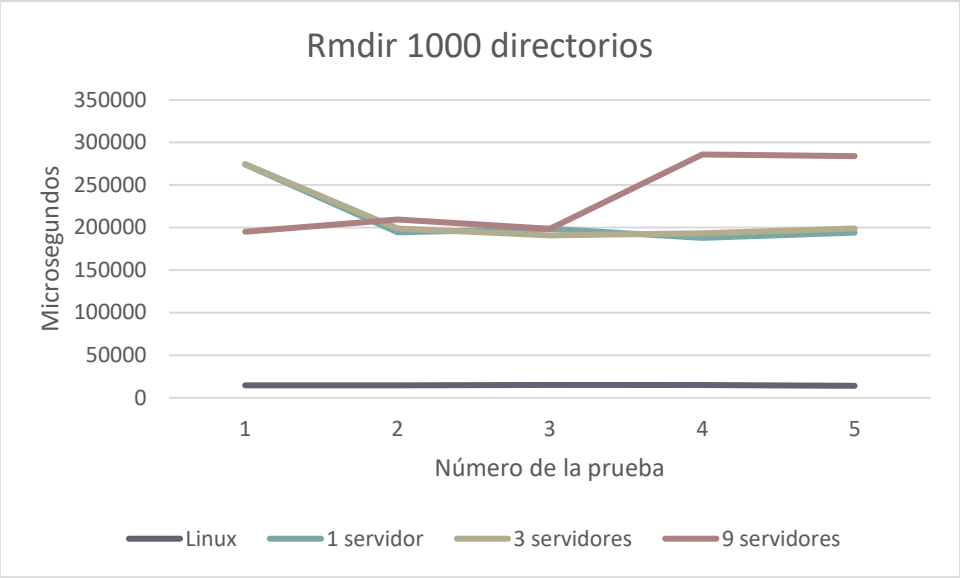


Ilustración 6-2. Prueba rmdir 1000 directorios

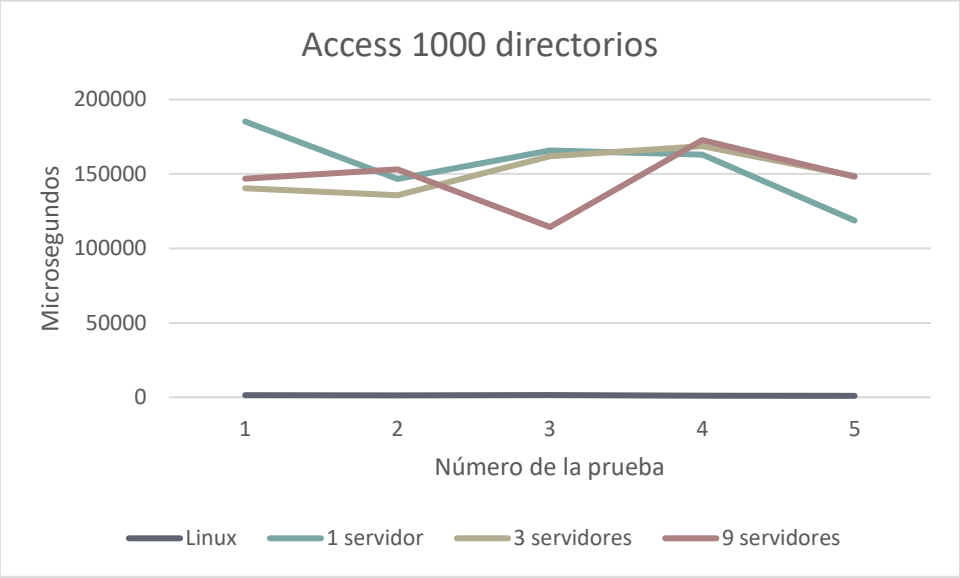


Ilustración 6-3. Prueba access 1000 directorios

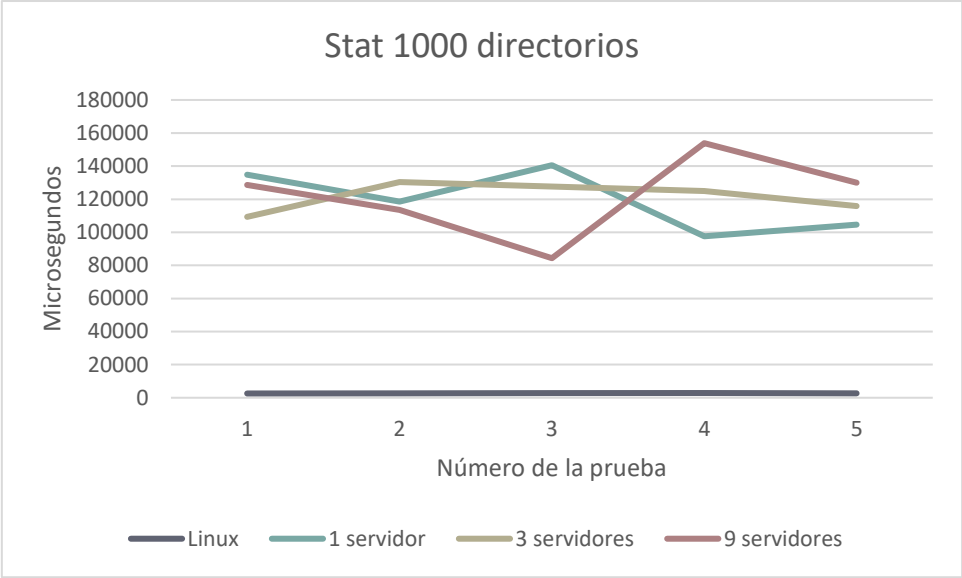


Ilustración 6-4. Prueba stat 1000 directorios

A partir de estas pruebas, es posible analizar los resultados y comprobar que las operaciones del sistema de ficheros desarrollado tardan más en realizarse, pero se debe de tener en cuenta que la escala son microsegundos. En realidad, se están comparando duraciones de 0.25 segundos del sistema de ficheros desarrollado con duraciones de 0.015 segundos del sistema de ficheros de Linux. Teniendo en cuenta que el sistema desarrollado ocurre a través de internet, se puede concluir que es un sistema rápido.

La segunda conclusión que se obtiene es que la operación realizada es indiferente, ya que una operación de un solo cliente siempre será atendida por un solo servidor de metadatos, así que las siguientes pruebas consisten en múltiples clientes al mismo tiempo.

Debido a dificultades técnicas para conectarse al clúster y hacer las pruebas en dicho clúster, a partir de este punto todas las pruebas expuestas serán en la máquina local. Consecuentemente, se observará una reducción de tiempos para las mismas tareas, la ejecución en el clúster suponía aproximadamente un aumento de entre 70 mil microsegundos y 250 mil microsegundos según las pruebas que dieron tiempo a realizar antes de perder el acceso.

A continuación, se exponen los tiempos para la creación de 2000 directorios, 1000 por cada cliente al mismo tiempo sobre un solo servidor de metadatos.

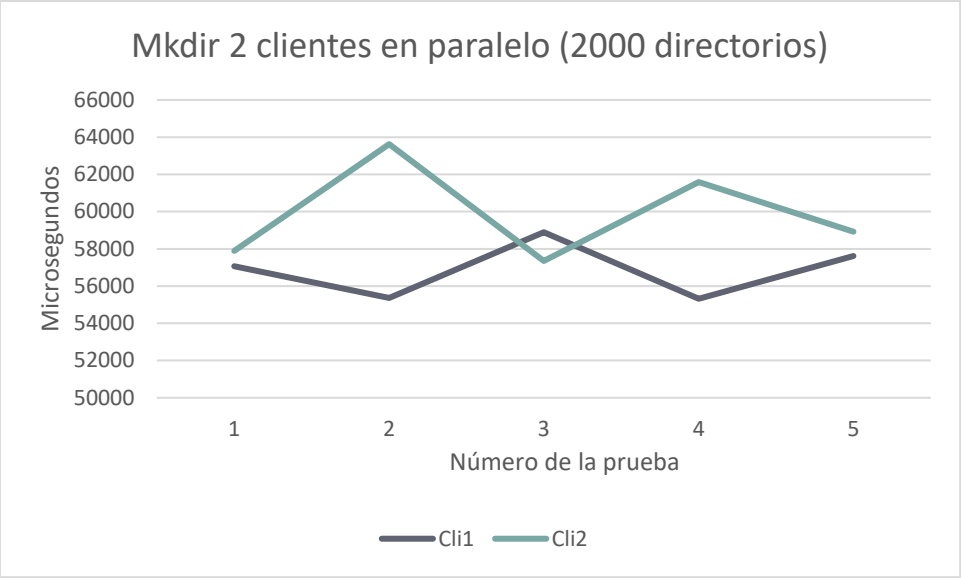


Ilustración 6-5. Mkdir de 2 clientes en paralelo

La siguiente prueba es, sobre las mismas circunstancias que la anterior, pero el borrado de 2000 directorios (los creados por la operación anterior).

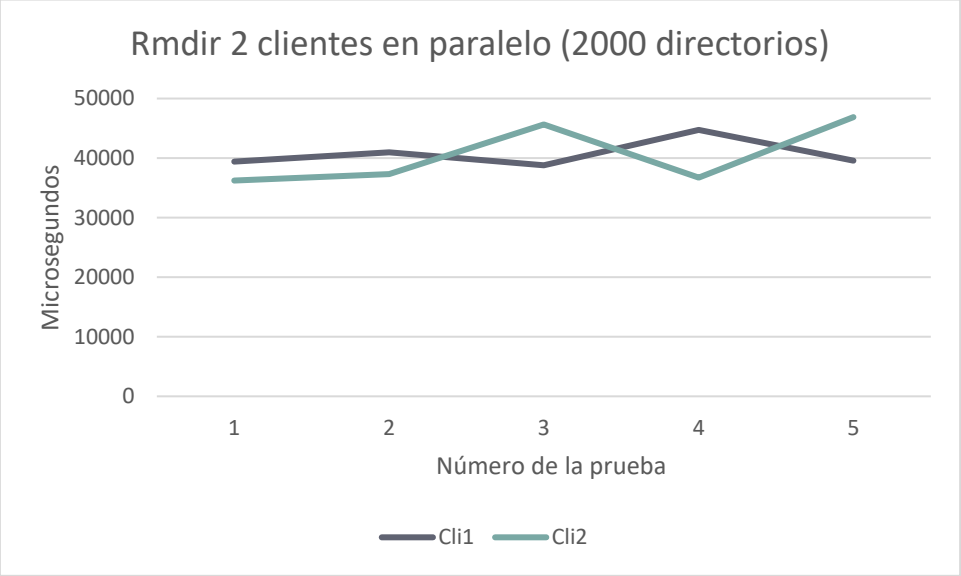


Ilustración 6-6. Rmdir de 2 clientes en paralelo

En ambas gráficas, se puede observar una tendencia de que cuando uno de los clientes tarda más, el otro necesariamente tarda menos en la misma ejecución, manteniendo una media prácticamente constante a lo largo de las ejecuciones.

A continuación, se amplía el número de clientes paralelos a 4 en la operación de access. Igual que en las pruebas anteriores, esto se realiza sobre un solo servidor de metadatos para evaluar la carga que puede soportar. El número total de directorios accedidos es de 4000.

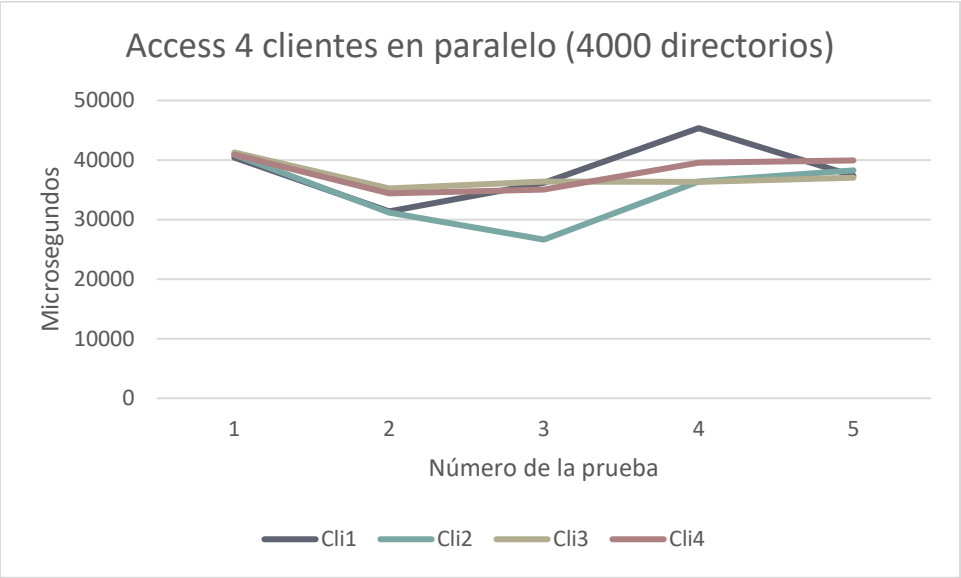


Ilustración 6-7. Access con 4 clientes en paralelo

Para concluir las pruebas de la sección de metadatos, se hace la prueba de stat para 8000 directorios, con 8 clientes en paralelo (ejecutados a la vez). Una vez más, esta carga se realiza toda sobre un solo servidor de metadatos.

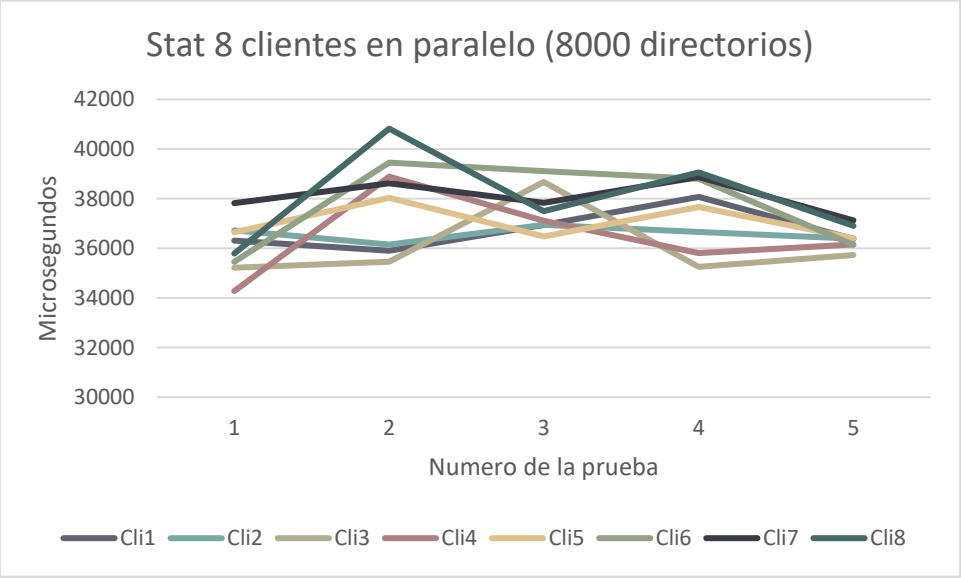


Ilustración 6-8. Stat 8 clientes en paralelo

Tanto en access como en stat, se puede observar la misma tendencia que en mkdir y rmdir, cuando algunos de los clientes tardan más, los demás tardan menos, resultando en una media que se aproxima a un tiempo constante.

Mediante estas pruebas sobre los metadatos, es posible deducir que el sistema es muy escalable, ya que apenas se ve afectado sobre una cantidad de hasta 8 clientes simultáneos sobre un solo servidor de metadatos.

6.2. Evaluación del sistema de datos

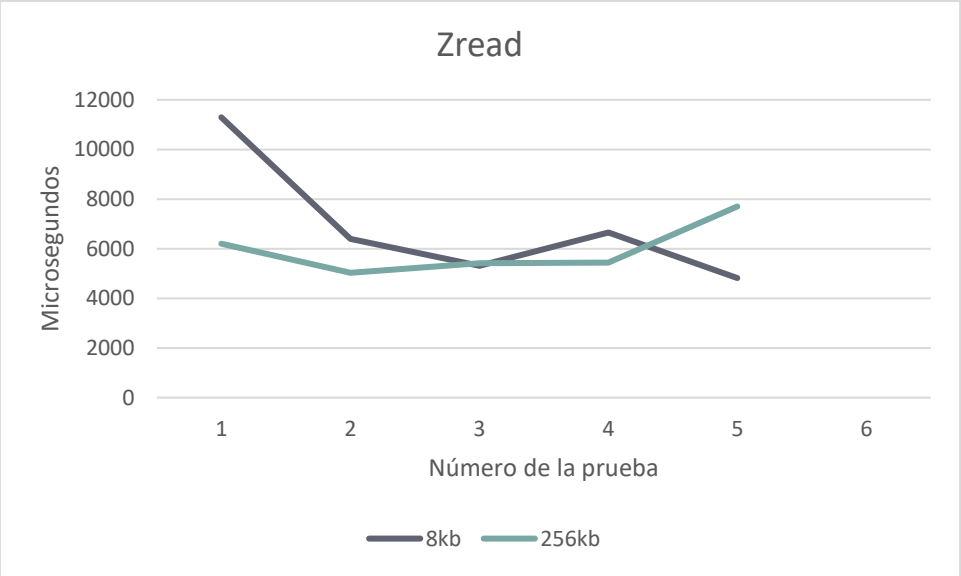


Ilustración 6-9. Prueba zread 8kb frente a 256kb

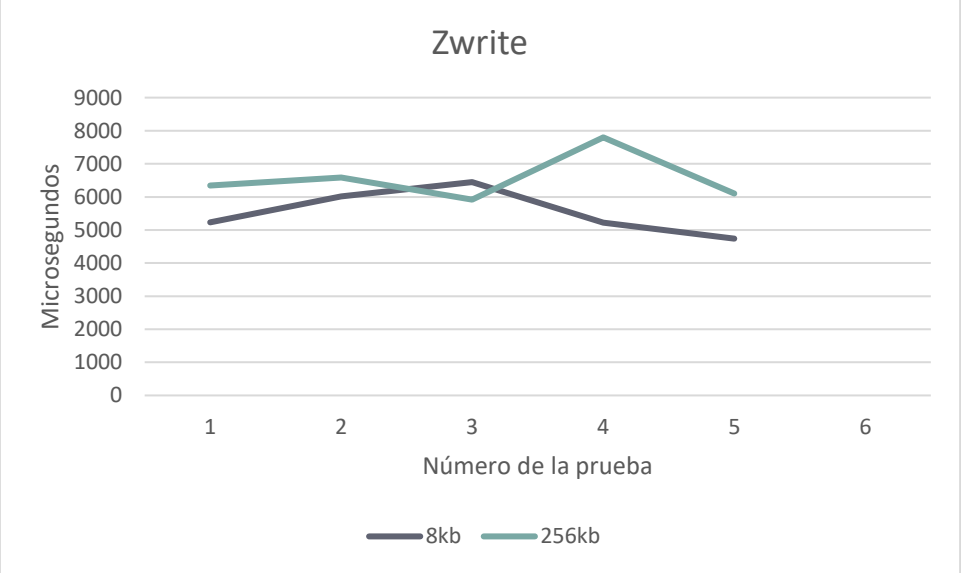


Ilustración 6-10. Prueba zwrite 8kb frente a 256kb

Las pruebas realizadas en el sistema de ficheros de Linux tardaron en el orden de 50 microsegundos las de 8kb y 800 microsegundos las de 256kb. Mediante las pruebas en local, con el sistema de datos desarrollado como prueba de concepto se obtiene una media de 6000 microsegundos en operaciones de lectura y 6000 microsegundos en pruebas de escritura.

Con estos tamaños de bloque, ni ZooKeeper ni los servidores de datos desarrollados experimentan una gran carga. De nuevo, si estas operaciones se hubiesen podido ejecutar en el clúster probablemente hubiese aumentado la duración de las operaciones entre 70 mil y 250 mil microsegundos adicionales (de forma estimada).

7. PLANIFICACIÓN Y PRESUPUESTO

En este apartado se explicará cuál ha sido la planificación anterior que se realizó antes de comenzar con el proyecto con unos tiempos estimados, después se incluirán unos diagramas de Gantt donde se verá cuál es el tiempo real que llevó completar el proyecto y la memoria.

Por otra parte, se incluirá un presupuesto donde se verá cual es el coste estimado que tiene el desarrollo de este sistema genérico de metadatos.

7.1. Planificación previa

La planificación inicial que se acordó con el tutor fue:

- Desarrollo de un script para descargar automáticamente ZooKeeper y descomprimirlo para tenerlo listo para ejecución. Tiempo estimado: 1 semana
- Descarga y toma de contacto con ZooKeeper en su versión implementada en C. Tiempo estimado: 1 semana
- Desarrollo de funciones relativas a sistemas de ficheros en estándar POSIX. Tiempo estimado: 3 meses
- Para poder desarrollar todas las funciones, las relativas a escritura y lectura necesitaban de un servidor de datos, por tanto, cuando llegase el momento hacía falta implementar un servidor de datos y unirlo al de metadatos mediante las funciones read y write. Tiempo estimado: 1 mes
- Desarrollo de un cliente que haga uso del sistema de ficheros completo. Tiempo estimado: 1 mes

7.2. Tiempo real en el que se realizó el proyecto

A continuación, se presenta una tabla donde se indican cada una de las actividades relacionadas con los números de semanas planificadas y reales desde la fecha de inicio del proyecto que fue el 8 de noviembre de 2017, hasta el final que fue el 1 de junio de 2018, un tiempo estimado de 7 meses y 3 semanas.

ACTIVIDAD	INICIO DEL PLAN	DURACIÓN DEL PLAN	INICIO REAL	DURACIÓN REAL	PORCENTAJE COMPLETADO
Script	1	1	1	1	100%
Toma de contacto ZooKeeper C	2	1	2	1	100%
Desarrollo de funciones para la API	3	12	3	24	100%
Desarrollo del servidor de datos	15	4	15	2	100%
Desarrollo del cliente	15	4	3	25	100%
Desarrollo de la memoria	24	8	24	8	100%

Tabla 37. Comparación entre la planificación y el tiempo real del proyecto

A partir de la tabla anterior, se representa un diagrama de Gantt con todas las tareas y el trabajo realizado en cada una de dichas tareas en cada una de las semanas.

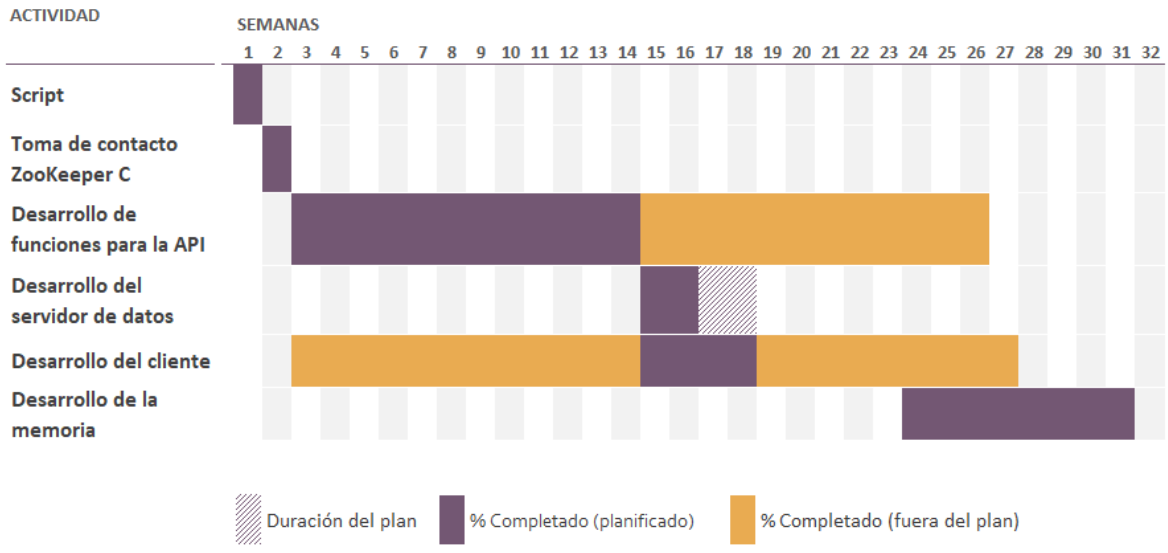


Ilustración 7-1. Diagrama de Gantt

Como podemos observar en el diagrama, las principales diferencias sobre la planificación previa son que el desarrollo de las funciones para la API llevó más tiempo del estimado inicialmente, se extendió al doble de semanas de las estimadas, concretamente

comprendió un período de 26 semanas o 6 meses. También observamos que el desarrollo del servidor de datos se completó en dos semanas cuando lo estimado fue un mes. El desarrollo del cliente fue paulatino desde el inicio del desarrollo de las funciones para la API, ya que mediante el cliente se hacían las pruebas necesarias para verificar que las funciones operaban correctamente.

7.3. Presupuesto

En base al tiempo real que ha durado el proyecto, se va a estimar en primer lugar un coste del personal, que consisten principalmente en el sueldo de un programador. Por otro lado, se necesitan estimar costes indirectos como costes de software utilizado, la electricidad e internet entre otros.

El sueldo medio de un programador en España dada la fuente [14] es de 25,000€ anuales, dado que el trabajo realizado ha sido en 7 meses y 3 semanas, redondeamos a 8 meses y hacemos la conversión de 25,000€ anuales a 8 meses obteniendo un salario de 16,667€. A esto hay que sumar la seguridad social, que dada la fuente [15] es un 28,30% sobre el sueldo que sobre 16,667€ es 4717€, si sumamos ambas resulta en unos costes directos de 21,384€.

En cuanto a los costes indirectos, se exponen en la siguiente tabla:

Categoría	Especificación	Coste por mes	Coste total
Software	Licencia Windows 10	-	Gratuito por la universidad
Software	Licencia Office 365	-	Gratuito por la universidad
Software	ZooKeeper	-	Software gratuito
Software	Linux	-	Software gratuito
Electricidad	Fuente [16]	70€	560€
Internet	Fibra óptica 500MB	80€	640€
		Total:	**1200€**

Tabla 38. Costes indirectos

8. CONCLUSIONES

En este apartado se escribirán las conclusiones sobre el proyecto que se ha desarrollado, así como sobre el proceso que se ha seguido para completar el proyecto. Por otro lado, se incluirán unas conclusiones personales y posibles trabajos futuros.

8.1. Conclusiones sobre el proyecto

El desarrollo de un sistema genérico de metadatos utilizando ZooKeeper y acoplado junto con un servidor de datos como prueba de concepto ha terminado siendo, en general, un éxito.

Exceptuando un problema respecto a la funcionalidad planeada desde un comienzo, el proyecto ha cumplido con las expectativas. El problema que se ha encontrado ha sido que ZooKeeper no soporta el renombrado de nodos de forma nativa, por tanto, la implementación de la API desarrollada no ha podido hacer una llamada de renombrado simplemente. Se ha terminado implementado un sistema de renombrado de ficheros o directorios vacíos, es decir, únicamente para los nodos hoja del sistema de ficheros.

Respecto a los aspectos positivos, se ha conseguido un sistema genérico de metadatos con un estándar POSIX, un cliente que puede consumir de ese sistema, así como una implementación de servidores de datos que sirve como prueba de concepto de que el sistema funciona en conjunto.

Sin duda, volvería a utilizar ZooKeeper para otras implementaciones que necesiten un manejo distribuido y paralelo sobre un sistema de metadatos. Además, aunque las funciones desarrolladas hacen llamadas a funciones que componen la mayoría de las disponibles de ZooKeeper, existen algunas operaciones interesantes relativas a "watchers" o supervisores que envían notificaciones ante modificación de nodos que se están consultado. Esta funcionalidad de "watchers" no se ha explorado y despierta el interés.

8.2. Conclusiones sobre el proceso

Gracias a la planificación inicial, considero que el proyecto ha tenido una estructura definida y clara. Durante todo el proceso de desarrollo se ha seguido de cerca la planificación, excepto en la cuestión del desarrollo de cliente. Considero que el desarrollo de cliente (que inicialmente estuvo planificado para el final) fue necesario desde el

principio del desarrollo para poder hacer un proceso de programación, revisión y pruebas de la API continuamente.

Respecto del servidor de datos, inicialmente se pensó en acoplar el sistema con un servidor de datos RPC ya desarrollado, pero al tener casi terminado el sistema de metadatos y debido al volumen del servidor de datos RPC, se pensó que sería mejor realizar un desarrollo de un servidor de datos propio que sirviese como prueba de concepto.

Como error personal considero que el desarrollo de algunos aspectos de la memoria como el de diseño e implantación se podrían haber hecho antes, me llevo la lección de que es útil documentar los avances según se realizan.

8.3. Conclusiones personales

Mediante la realización de este proyecto, se ha profundizado en el ámbito de sistemas de ficheros tradicionales, así como en los sistemas de ficheros paralelos y distribuidos, haciendo especial énfasis en la parte de metadatos.

Los conocimientos existentes y los adquiridos respecto de los sistemas de ficheros tradicionales han servido para contrastar metodologías y protocolos respecto del almacenamiento de información.

Sin duda, ha aumentado el conocimiento de programación en C y la habilidad de consumir de una API existente (en este caso la de ZooKeeper). Cabe destacar que después de haber trabajado con ZooKeeper se valora y aprecia su utilidad mucho más que cuando se escuchó por primera vez de su existencia y funcionalidad.

Se ha aprendido a resolver problemas en entornos distribuidos como el de apertura de ficheros y directorios, cuyas estructuras tienen que extraerse del servidor de metadatos, pero almacenarse en el cliente.

Se ha mejorado la percepción y el conocimiento general de los sistemas de metadatos distribuidos, cuál es su arquitectura y cómo se relacionan los componentes. También se ha aprendido la importante lección de que un solo servidor de metadatos no es suficiente, como se ha visto en el estado del arte, y que su importancia es tan grande como la de los servidores de datos.

También se han refrescado los conocimientos sobre servidores basados en sockets mediante el desarrollo de servidores de datos como prueba de concepto. Se ha profundizado en el estándar POSIX, lo que mejora los conocimientos existentes sobre cualquier aplicación o sistema que utilice dicho estándar, especialmente si son funciones relacionadas con sistemas de ficheros.

En resumen, quedo satisfecho por haber realizado este proyecto ya que considero que he aprendido distintos conceptos que me han abierto la mente. Ahora cuando pienso en sistemas de ficheros o cualquier aplicación distribuida, me puedo imaginar cómo funciona, mientras que antes tenía más dificultades para hacerlo.

8.4. Trabajos futuros

Los proyectos que se podrían realizar para mejorar éste pueden ser respecto a varios componentes. La principal mejora sería acoplar algún otro sistema de datos que sea más eficiente en el almacenaje de datos.

Inicialmente el sistema desarrollado no iba a tener acoplado un sistema de datos que sirviese como prueba de concepto de que el sistema funciona en conjunto, sino que iba tener un servidor de datos mediante llamadas RPC, por ello un posible proyecto futuro podría ser sustituir el sistema de datos actual por uno ya hecho que utilice llamadas RPC. Para ello es posible que haga falta modificar el código de las función read y write mínimamente para garantizar un protocolo que ambas partes entiendan.

Una posible mejora sobre el servicio de metadatos es establecer unos permisos más claros respecto de los permisos de grupo y permisos para el resto de los usuarios. En la implementación actual, se hace hincapié en los permisos que tiene el propietario del fichero, para el resto de los usuarios que no son propietarios el acceso está completamente denegado.

Otra posible mejora es realizar una especificación de errores más detallada y que obedezca mejor el estándar POSIX, devolviendo correctamente el “ERRNO” y cubriendo una mayor posibilidad de errores.

Existe un mecanismo de autenticación en ZooKeeper mediante unas credenciales, similares a usuario y contraseña, al que se podría sacar partido para aumentar la seguridad y privacidad del sistema de metadatos.

Una mejora menos importante que quizá pueda resultar útil a algún tipo de cliente es implementar enlaces simbólicos, que es un mecanismo similar a los de accesos directos, de forma que se puedan referenciar a otros archivos o directorios desde cualquier parte del sistema de ficheros.

9. APÉNDICES

Si no es posible compilar según lo indicado en el apartado de implantación, es posible que el fallo se deba a que la compilación con make de las librerías por defecto de ZooKeeper fallan. Para arreglar el fallo hace falta modificar el fichero configure.ac y añadir las siguientes líneas:

AC_CONFIG_MACRO_DIRS([/usr/local/aclocal-1.15/])

m4_ifdef([AM_PROG_AR]

Se añaden justo debajo de AC_CONFIG_SRCDIR([src/zookeeper.c]).

Si se especifica la ruta /usr/local/aclocal-1.15/, el fichero cppunit.m4 debe de estar localizado en esa ruta, si el fichero cppunit.m4 está localizado en otro directorio, se debe de indicar sustituyendo esta ruta por la apropiada.

También hay que cambiar la línea:

AM_INIT_AUTOMAKE([-Wall foreign])

Por la línea:

AM_INIT_AUTOMAKE([-Wall foreign subdir-objects])

Después de estos cambios es posible hacer make sin que falle y se puede proceder a compilar las librerías desarrolladas mediante libtool.

10. BIBLIOGRAFÍA

[1]: Lustre File System 2018 [cited 01/05 2018]. Available from http://lustre.org

[2]: Carns H., Philip, Walter Ligon B., Robert Ross B., and Rajeev Thakur. 2000. PVFS: A Parallel File System for Linux Clusters. Available from: http://www.mcs.anl.gov/~thakur/papers/pvfs.pdf

[3]: Osadzinski, Alex. 1988. The network file system (NFS). Abstract available from: https://dl.acm.org/citation.cfm?id=54202

[4]: Ren, Kai, Qing Zheng, Swapnil Patil, and Garth Gibson. 2014. Scaling File System Metadata Performance with Stateless Caching and Bulk Insertion. Available from: https://pdfs.semanticscholar.org/265d/a96369ea4988aac2fd98a69e66f553cc07cf.pdf

[5]: Burrows, Mike. 2006. The Chubby lock service for loosely-coupled distributed systems. Available from: https://ai.google/research/pubs/pub27897

[6]: Apache Software Foundation. 2018 [cited 05/01 2018]. Available from: https://hadoop.apache.org/docs/current/hadoop-project-dist/hadoop-hdfs/HdfsDesign.html

[7]: Apache Software Foundation. ZooKeeper 3.4 Documentation. 2018 [cited 05/01 2018]. Available from: https://zookeeper.apache.org/doc/current/zookeeperOver.html

[8]: OpenAFS. 2018 [cited 05/01 2018]. Available from: https://www.openafs.org

[9]: Ghemawat, Sajnay, Howard Gobioff, and Shun-Tak Leung. 2003. The Google File System. Available From: https://ai.google/research/pubs/pub51

[10]: Applications and organizations using ZooKeeper 2014 [cited 05/01 2018]. Available from: https://cwiki.apache.org/confluence/display/ZOOKEEPER/PoweredBy

[11]: Zimmerman, Jordan. Introducing Curator—The Netflix ZooKeeper Library. 2011 [cited 05/01 2018]. Available from: https://medium.com/netflix-techblog/introducing-curator-the-netflix-zookeeper-library-c814d3f4917c

[12]: Apache Software Foundation. ZooKeeper. 2018 [cited 05/01 2018]. Available from: https://zookeeper.apache.org

[13]: GNU General Public License. 2016 [cited 05/01 2018]. Available from: https://www.gnu.org/licenses/gpl-3.0.en.html

[14]: https://www.glassdoor.com/Salaries/spain-software-engineer-salary-SRCH_IL.0,5_IN219_KO6,23.htm

[15]: Bases y tipos de cotización 2018. 2018 [cited 05/01 2018]. Available from: http://www.seg-social.es/Internet_1/Trabajadores/CotizacionRecaudaci10777/Basesytiposdecotiza36537/index.htm

[16]: Precio de la luz por horas. 2018 [cited 05/01 2018]. Available from: https://tarifaluzhora.es/